Proceedings

26th Annual Computer Security Applications Conference

ACSAC 2010

Proceedings

26th Annual Computer Security Applications Conference

Austin, Texas, USA
6–10 December 2010

Sponsored by
Applied Computer Security Associates

The Association for Computing Machinery
2 Penn Plaza, Suite 701
New York, New York 10121-0701

ACM ISBN: 978-1-4503-0133-6

Editorial production by Christoph Schuba
Published by ACM, Inc. within the ACM International Conference Proceedings Series

Table of Contents

Vulnerability Assessment of Embedded Devices

Classic Paper I

Invited Essayist

Botnets

Email, E-Commerce, and Web 2.0

Hardware-Assisted Security

Security Protocols and Portable Storage

Model Checking and Vulnerability Analysis

Classic Paper II

Intrusion Detection and Live Forensics

Distributed Systems and Operating Systems

Mobile and Wireless

Security Engineering and Management

Message from the Conference Chair

Welcome to the 26th Annual Computer Security Applications Conference (ACSAC). ACSAC is one of the premiere security conferences, unique in its focus on applied security. Our program is further strengthened through the consistent participation of attendees from both government and industry in addition to academia, resulting in stimulating conversations that help drive cybersecurity research forward.

We have a terrific program planned for this year! Our technical program features papers across a broad range of topics, such as social networks, botnets, and security engineering. I would like to thank our program chair, Michael Franz, and co-chair, John McDermott, for putting together an excellent program.

Thanks to Deb Frincke and Kevin Butler we have an exciting selection of guest speakers, including Doug Maughan from the Science and Technology Directorate of the Department of Homeland Security as our Distinguished Practitioner and Tom Longstaff from the Applied Information Science Department of the Johns Hopkins University Applied Physics Laboratory as our Invited Essayist. We are also excited to have Giovanni Vigna from UC Santa Barbara and William Cheswick from AT&T as our classic papers authors.

In addition to speakers and technical papers, we also have a wonderful selection of panels and case studies, including topics as diverse as cloud security, security economics, supply chain risk management, and the federal cyber security research agenda. I would like to thank Hongxia Jin and Steve Rome for their work on putting together such a great selection of panels and case studies.

Continuing due to its success last year, we have a fourth track for conference attendees that will focus on FISMA training for government employees. I would like to thank Marshall Abrams and Ron Ross for organizing this session.

We will again have our popular works-in-progress (WiP) talks this year together with a poster session. Thanks to Charles Payne (WiPs) and Ben Kuperman and James Early (posters) for their hard work on putting together these sessions.

New this year, we will be hosting a career night in conjunction with the WiPs and poster session. Many thanks go to Ben Cook and Kathyrn Hanselmann for making this session a success!

Thanks also go to Daniel Faigin for organizing a terrific line-up of tutorials and to Harvey Rubinovitz for putting together the conference workshop. Both will be held before the conference itself, and feature topics such as usable security and the state of the practice in intrusion detection. The workshop this year covers the governance issues behind information technology and policy.

Without the generous help of a large number of volunteers, this conference would not be possible. In addition to the people I've mentioned above, I would like to thank Art Friedman, Lillian Røstad, Mike Collins, Ken Shotting, Kevin Butler, Ed Schneider, Christoph Schuba, Kristin Steen, Jay Kahn, and Robert Zakon for all of their hard work and support.

And a special thanks goes to Jeremy Epstein for all of his work making arrangements with the hotel, and for his dedication to making ACSAC a success.

I hope that you enjoy the conference - we have an amazing program this year in a location known for great food and great music, and I look forward to meeting you there!

Carrie Gates, ACSAC 2010 Conference Chair

Message from the Program Chairs

Welcome to the 26th Annual Computer Security Applications Conference. We are happy to present a formidable technical program that is the work of many volunteers. We would like to thank all of these volunteers for their contributions to ACSAC 2010.

First, thanks go to the authors. We received a total of 239 submissions this year. After some papers were rejected on formal grounds or retracted by their authors, a total of 227 papers entered the reviewing phase, a new record for ACSAC.

Second, our sincere gratitude goes to the Program Committee, who gave extra time to review the unexpectedly large number of submissions. Every paper was initially reviewed by two PC members. In this first reviewing round, 132 papers received at least one score of "weak accept" or better and thereby proceeded to the second stage. Additional reviews were solicited for all 132 papers. Of these, a total of 77 papers received sufficiently high scores that they were ultimately discussed at the PC meeting.

An in-person PC meeting was held on Saturday, August 14th at SRI's offices in Arlington, Virginia. We are very grateful to Jeremy Epstein and SRI for hosting this all-day meeting. During the meeting, the PC selected 29 papers for immediate acceptance and an additional 10 papers for conditional acceptance subject to shepherding by members of the PC.

Reviews were double-blinded throughout: papers were submitted without author names and affiliations and PC members never learned the identity of submitters until the reviewing process was complete. Throughout the reviewing process, the two PC Chairs were the only people who knew which paper was authored by whom. We took great care to manage potential conflicts, since in our double-blind process, reviewers were often not even aware of such conflicts.

We are happy to report that all of the conditionally accepted papers made it into the final program. Thanks go to the authors and to the shepherds for the time they collaboratively invested in improving these papers for ACSAC 2010.

And finally, we thank the larger ACSAC community for your continuing support. Whether you are attending ACSAC in person this year or reading these proceedings elsewhere, we hope that you will find these papers interesting, inspiring, and relevant. Enjoy!

Michael Franz, ACSAC 2010 Program Chair

John McDermott, ACSAC 2010 Program Co-Chair

Conference Committee

Carrie Gates, CA Labs (Conference Chair)
Michael Franz, University of California, Irvine (Program Chair)
John McDermott, Naval Research Lab (Program Co-Chair)
Christoph Schuba, Oracle Corporation (Multimedia/Proceedings)
Daniel Faigin, Aerospace Corporation (Tutorials Chair)
Steve Rome, Booz Allen Hamilton (Case Studies Chair)
Ken Shotting, DoD (Case Studies Co-Chair)
Hongxia Jin, IBM (Panels Chair)
Art Friedman, NSA (Registration Chair)
Benjamin Kuperman, Oberlin College (Poster Chair)
James P. Early, State University of New York at Oswego (Poster Co-Chair)
Kevin Butler, University of Oregon (Publicity Chair)
Mike Collins, RedJack (Sponsorship Chair)
Charlie Payne, Adventium Labs (Works in Progress Chair)
Harvey H. Rubinovitz, MITRE (Workshop Chair)
Marshall Abrams, MITRE (FISMA Coordination Chair)
Jeremy Epstein, SRI International (Local Arrangements Chair)
Krstin Steen, Sandia National Lab (Local Arrangements Co-Chair)
Lillian Røstad, Norwegian University of Science and Technology (Student Awards)
Ben Cook, Sandia National Lab (Student Outreach Chair)
Deb Frincke, Pacific Northwest Lab (Guest Speaker Liaison Chair)
Kevin Butler, University of Oregon (Guest Speaker Liaison Co-Chair)
Ed Schneider, Institute for Defense Analyses (Treasurer)
Dan Thomsen, SIFT (Knowledge Coordinator)
Jay Kahn, MITRE (ACSA Communications Chair)
Cristina Serban, AT&T (Conference Chair Emerita)
Robert H'obbes' Zakon, Zakon Group (Web Advisor)

ACSAC Steering Committee

Marshall Abrams, The MITRE Corporation
Jeremy Epstein, SRI International
Daniel Faigin, The Aerospace Corporation
Ann Marmor-Squires, The Sq Group
Steve Rome, Booz Allen Hamilton
Ron Ross, National Institute of Standards
Christoph Schuba, Oracle Corporation
Cristina Serban, AT&T
Dan Thomsen, Cyber Defense Agency LLC

Program Committee

Michael Franz, University of California, Irvine (Program Chair)
John McDermott, Naval Research Lab (Program Co-Chair)
Vijay Atluri, Rutgers University
Tuomas Aura, Microsoft Research
Lee Badger, U.S. National Institute of Standards and Technology (NIST)
Elisa Bertino, Purdue University
Konstantin Beznosov, University of British Columbia
Matt Bishop, University of California, Davis
Sjdan Capcun, ETH Zurich
Fred Chong, University of California, Santa Barbara
Christian Collberg, University of Arizona
Marc Dacier, Symantec Corporation
Mary Denz, U.S. Air Force Research Laboratory
Sven Dietrich, Stevens Institute of Technology
Jeremy Epstein, SRI International
David Evans, University of Virginia
Richard Ford, Florida Institute of Technology
Tyrone Grandison, IBM Almaden Research Center
Steven Greenwald, Independent Security Advisor
Cynthia Irvine, U.S. Naval Postgraduate School
Trent Jaeger, Pennsylvania State University
Hongxia Jin, IBM Almaden Research Center
Michiharu Kudoh, IBM Tokyo Research Laboratory
Michael Locasto, University of Calgary
Patrick McDaniel, Pennsylvania State University
Peng Ning, North Carolina State University
Charles Payne, Adventium Labs
Andreas Pfitzmann, Technische Universität Dresden
Christian Probst, Technical University of Denmark (DTU)
Lillian Røstad, Norwegian University of Science and Technology
Reiner Sailer, IBM T.J. Watson Research Center
Pierangela Samarati, University of Milan
Christoph Schuba, Oracle Corporation
R. Sekar, Stony Brook University
Cristina Serban, AT&T
Frederick Sheldon, Oak Ridge National Laboratory
Brian Snow, Independent Security Advisor
Anil Somayaji, Carleton University
Angelos Stavrou, George Mason University
Bhavani Thuraisingham, University of Texas, Dallas
Patrick Traynor, Georgia Institute of Technology
Venkat Venkatakrishnan, University of Illinois at Chicago

External Review Committee

William Allen, Florida Institute of Technology
Kevin Butler, University of Oregon
Sunoh Choi, Purdue University
William Enck, Pennsylvania State University
Vinod Ganapathy, Rutgers University
Ashish Kamra, Purdue University
Ashish Kundu, Purdue University
Liam Mayron, Harris Corp.
Thomas Moyer, Pennsylvania State University
Albert Noll, ETH Zurich
Machigar Ongtang, Pennsylvania State University
Christian Wimmer, University of California, Irvine

Additional Reviewers

Scott Adams, Claudio Ardagna, Prithvi Bisht, Jeff Browne, Jonathan Burket, Joe Carozzoni, Peter Chapman, Sabrina De Capitani di Vimercati, Sara Foresti, Karthik Thotta Ganesh, Jason Gionta, Kalpana Gondi, Lakshmi Gowri, Steven Harp, Kirstie Hawkey, Amir Herzberg, Yan Huang, Ted Huffmire, James Hughes, Pooya Jaferian, Quan Jia, Xuxian Jiang, Gunes Kayacik, Pranab Kini, Herb Klumpe, Loukas Lazos, Hsien-Hsin S. Lee, Bita Masloom, Steve McLaughlin, Sara Motiee, Ildar Muslukhov, Divya Muthukumaran, Mohammad Nikseresht, Richard O'Brien, Fahimeh Raja, Sandra Rueda, Heba Saadeldeen, Joshua Schiffman, Ferdinand Schober, Brian Sessler, Haya Shulman, Andreas Sotirakopoulos, Anna Squicciarini, San-Tsai Sun, Mike Ter Louw, Mohit Tiwari, Sven Türpe, Alfonso Valdes, Praveen Venkatachari, Jiang Wang, Zhaohui Wang, Hassan Wassel, Mark Williams, Yu Yao, Michelle Zhou, Yuchen Zhou

Tutorial Reviewers

Daniel P. Faigin, The Aerospace Corporation, (ACSAC Tutorial Chair)
M. Patrick Collins, RedJack LLC
Patricia Daggett, EMC Corporation
Deborah D. Downs, The Aerospace Corporation
Tyrone W A Grandison, IBM Almaden Reserch Center
Jay Kahn, Retired
Donna Mitchell, Lockheed Martin Corporation
Charles Payne, Adventium Labs
W. Warren Pearce, Northrop Grumman Corp.
Marco Ramilli, University of Bologna, Italy
Max Robinson, The Aerospace Corporation
Steven Rome, Booz Allen Hamilton
Michelle Ruppel, Saffire Systems
Harvey Rubinovitz, The MITRE Corporation
Cristina Serban, AT&T
Paul Streander, The Aerospace Corporation
Danielle Weigold, Lockheed Martin
Simon Wiseman, Deep-Secure Ltd

Message from the Sponsor

Applied Computer Security Associates

ACSA had its genesis in the first Aerospace Computer Security Applications Conference in 1985. That conference was a success and evolved into the Annual Computer Security Applications Conference (ACSAC). ACSA was incorporated in 1987 as a non-profit association of computer security professionals who have a common goal of improving the understanding, theory, and practice of computer security. ACSA continues to be the primary sponsor of the annual conference.

In 1989, ACSA began the Distinguished Practitioner Series at the annual conference. Each year, an outstanding computer security professional is invited to present a lecture of current topical interest to the security community.

In 1991, ACSAC began the Best Paper by a Student Award, presented at the Annual conference. This award is intended to encourage active student participation in the conference. The award winning student author receives an honorarium and conference expenses. Additionally, our Student Conferenceship program assists selected students in attending the Conference by paying for the conference fee. Applicants must be undergraduate or graduate students, nominated by a faculty member at an accredited university or school, and show the need for financial assistance to attend this conference.

An annual prize for the Outstanding Paper has been established for the Annual Computer Security Applications Conference. The winning author receives a plaque and an honorarium. The award is based on both the written and oral presentations.

ACSA initiated the Marshall D. Abrams Invited Essay in 2000 to stimulate development of provocative and stimulating reading material for students of Information Security, thereby forming a set of Invited Essays. Each year's Invited Essay addresses an important topic in Information Security not adequately covered by the existing literature.

This year's ACSAC continues the Classic Papers feature begun in 2001. The classic papers are updates of some of the seminal works in the field of Information Security that reflect developments in the research community and industry since their original publication. Each presentation also considers how these classical security results will impact us in the years to come.

ACSA continues to be committed to serving the security community by finding additional approaches for encouraging and facilitating dialogue and technical interchange. In previous years, ACSA has sponsored small workshops to explore various topics in Computer Security (in 2000, the Workshop on Innovations in Strong Access Control; in 2001, the Workshop on Information Security System Rating and Ranking; in 2002, the Workshop on Application of Engineering Principles to System Security Design). In 2003, ACSA became the sponsor of the already established New Security Paradigms Workshop (NSPW). This year, ACSA is excited to welcome the Layered Assurance Workshop as an affiliated ACSA activity. ACSA also maintains a Classic Papers Bookshelf that preserves seminal works in the field and a web site focusing on Strong Access Control/Multi-Level Security.

ACSA is always interested in suggestions from interested professionals and computer security professional organizations on other ways to achieve its objectives of encouraging and facilitating dialogue and technical interchange. For more information on ACSA and its activities, please visit http://www.acsac.org/acsa

To learn more about the conference, visit the ACSAC web page at http://www.acsac.org

ACSA Members

Marshall Abrams, The MITRE Corporation (ACSA Founder and Assistant Treasurer)
Jeremy Epstein,SRI International (ACSA Vice President)
Daniel Faigin, The Aerospace Corporation (ACSA Secretary)
Ann Marmor-Squires, The Sq Group
Steve Rome, Booz Allen Hamilton (ACSA President)
Harvey Rubinovitz, The MITRE Corporation (ACSA Treasurer)
Cristina Serban, AT&T
Mary Ellen Zurko, IBM Corporation

Detecting Spammers on Social Networks

Gianluca Stringhini
University of California, Santa Barbara
gianluca@cs.ucsb.edu

Christopher Kruegel
University of California, Santa Barbara
chris@cs.ucsb.edu

Giovanni Vigna
University of California, Santa Barbara
vigna@cs.ucsb.edu

ABSTRACT

Social networking has become a popular way for users to meet and interact online. Users spend a significant amount of time on popular social network platforms (such as Facebook, MySpace, or Twitter), storing and sharing a wealth of personal information. This information, as well as the possibility of contacting thousands of users, also attracts the interest of cybercriminals. For example, cybercriminals might exploit the implicit trust relationships between users in order to lure victims to malicious websites. As another example, cybercriminals might find personal information valuable for identity theft or to drive targeted spam campaigns.

In this paper, we analyze to which extent spam has entered social networks. More precisely, we analyze how spammers who target social networking sites operate. To collect the data about spamming activity, we created a large and diverse set of "honey-profiles" on three large social networking sites, and logged the kind of contacts and messages that they received. We then analyzed the collected data and identified anomalous behavior of users who contacted our profiles. Based on the analysis of this behavior, we developed techniques to detect spammers in social networks, and we aggregated their messages in large spam campaigns. Our results show that it is possible to automatically identify the accounts used by spammers, and our analysis was used for take-down efforts in a real-world social network. More precisely, during this study, we collaborated with Twitter and correctly detected and deleted 15,857 spam profiles.

1. INTRODUCTION

Over the last few years, social networking sites have become one of the main ways for users to keep track and communicate with their friends online. Sites such as Facebook, MySpace, and Twitter are consistently among the top 20 most-viewed web sites of the Internet. Moreover, statistics show that, on average, users spend more time on popular social networking sites than on any other site [1]. Most social networks provide mobile platforms that allow users to access their services from mobile phones, making the access to these sites ubiquitous.

The tremendous increase in popularity of social networking sites allows them to collect a huge amount of personal information about the users, their friends, and their habits. Unfortunately, this wealth of information, as well as the ease with which one can reach many users, also attracted the interest of malicious parties. In particular, spammers are always looking for ways to reach new victims with their unsolicited messages. This is shown by a market survey about the user perception of spam over social networks, which shows that, in 2008, 83% of the users of social networks have received at least one unwanted friend request or message [16].

From a security point of view, social networks have unique characteristics. First, information access and interaction is based on trust. Users typically share a substantial amount of personal information with their friends. This information may be public or not. If it is not public, access to it is regulated by a network of trust. In this case, a user allows only her friends to view the information regarding herself. Unfortunately, social networking sites do not provide strong authentication mechanisms, and it is easy to impersonate a user and sneak into a person's network of trust [15]. Moreover, it often happens that users, to gain popularity, accept any friendship request they receive, exposing their personal information to unknown people. In other cases, such as MySpace, the information displayed on a user's page is public by design. Therefore, anyone can access it, friend or not. Networks of trust are important from a security point of view, because they are often the only mechanism that protects users from being contacted by unwanted entities.

Another important characteristic of social networks is the different levels of user awareness with respect to threats. While most users have become aware of the common threats that affect the Internet, such as e-mail spam and phishing, they usually do not show an adequate understanding of the threats hidden in social networks. For example, a previous study showed that 45% of users on a social networking site readily click on links posted by their "friend" accounts, even if they do not know that person in real life [10]. This behavior might be abused by spammers who want to advertise web sites, and might be particularly harmful to users if spam messages contain links to malicious pages.

Even though social networks have raised the attention of researchers, the problem of spam is still not well understood. This paper presents the results of a year-long study of spam activity in social networks. The main contributions of this paper are the following:

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

- We created a set of honeynet accounts (honey-profiles) on three major social networks, and we logged all the activity (malicious or not) these accounts were able to observe over a one-year period for Facebook and an eleven-month period for Twitter and MySpace.
- We investigate how spammers are using social networks, and we examine the effectiveness of the countermeasures that are taken by the major social network portals to prevent spamming on their platforms.
- We identify characteristics that allow us to detect spammers in a social network.
- We built a tool to detect spammers, and used it on a Twitter and Facebook dataset. We obtained some promising results. In particular, we correctly detected 15,857 on Twitter, and after our submission to the Twitter spam team, these accounts were suspended.

2. BACKGROUND AND RELATED WORK

Social networks offer a way for users to keep track of their friends and communicate with them. This network of trust typically regulates which personal information is visible to whom. In our work, we looked at the different ways in which social networks manage the network of trust and the visibility of information between users. This is important because the nature of the network of trust provides spammers with different options for sending spam messages, learning information about their victims, or befriending someone (to appear trustworthy and make it more difficult to be detected as a spammer).

2.1 The Facebook Social Network

Facebook is currently the largest social network on the Internet. On their website, the Facebook administrators claim to have more than 400 million active users all over the world, with over 2 billion media items (videos and pictures) shared every week [3].

Usually, user profiles are not public, and the right to view a user's page is granted only after having established a relationship of trust (paraphrasing the Facebook terminology, *becoming friends*) with the user. When a user A wants to become friend with another user B, the platform first sends a request to B, who has to acknowledge that she knows A. When B confirms the request, a friendship connection with A is established. However, the users' perception of Facebook friendship is different from their perception of a relationship in real life. Most of the time, Facebook users accept friendship requests from persons they barely know, while in real life, the person asking to be friend would undergo more scrutiny.

In the past, most Facebook users were grouped in *networks*, where people coming from a certain country, town, or school could find their neighbors or peers. The default privacy setting for Facebook was to allow all people in the same network to view each other's profiles. Thus, a malicious user could join a large network to crawl data from the users on that network. This data allows an adversary to carry out targeted attacks. For example, a spammer could run a campaign that targets only those users whose profiles have certain characteristics (e.g., gender, age, interests) and who, therefore, might be more responsive to that campaign. For this reason, Facebook deprecated geographic networks in October 2009. School and company networks are still available, but their security is better, since to join one of these networks, a user has to provide a valid e-mail address from that institution (e.g., a university e-mail address).

2.2 The MySpace Social Network

MySpace was the first social network to gain significant popularity among Internet users. The basic idea of this network is to provide each user with a web page, which the user can then personalize with information about herself and her interests. Even though MySpace has also the concept of "friendship," like Facebook, MySpace pages are public by default. Therefore, it is easier for a malicious user to obtain sensitive information about a user on MySpace than on Facebook. Users might be profiled by gender, age, or nationality, and an aimed spam campaign could target a specific group of users to enhance its effectiveness.

MySpace used to be the largest social network on the Internet. Although it is steadily losing users, who are mainly moving to Facebook [2], it remains the third most visited site of its kind on the Internet.

2.3 The Twitter Social Network

Twitter is a much simpler social network than Facebook and MySpace. It is designed as a microblogging platform, where users send short text messages (i.e., *tweets*) that appear on their friends' pages. Unlike Facebook and MySpace, no personal information is shown on Twitter pages by default. Users are identified only by a username and, optionally, by a real name. To profile a user, it is possible to analyze the tweets she sends, and the feeds to which she is subscribed. However, this is significantly more difficult than on the other social networks.

A Twitter user can start "following" another user. As a consequence, she receives the user's tweets on her own page. The user who is "followed" can, if she wants, follow the other one back. Tweets can be grouped by *hashtags*, which are popular words, beginning with a "#" character. This allows users to efficiently search who is posting topics of interest at a certain time. When a user likes someone's tweet, he can decide to *retweet* it. As a result, that message is shown to all her followers. By default, profiles on Twitter are public, but a user can decide to protect her profile. By doing that, anyone wanting to follow the user needs her permission. According to the same statistics, Twitter is the social network that has the fastest growing rate on the Internet. During the last year, it reported a 660% increase in visits [2].

2.4 Related Work

The success of social networks has attracted the attention of security researchers. Since social networks are strongly based on the notion of a network of trust, the exploitation of this trust might lead to significant consequences. In 2008, a Sophos experiment showed that 41% of the Facebook users who were contacted acknowledged a friend request from a random person [8]. Bilge et al. [10] show that after an attacker has entered the network of trust of a victim, the victim will likely click on any link contained in the messages posted, irrespective of whether she knows the attacker in real life or not. Another interesting finding was reported by Jagatic et al. [13]. The authors found that phishing attempts are more likely to succeed if the attacker uses stolen information from victims' friends in social networks to craft

their phishing emails. There are also botnets that target social networks, such as koobface [9].

Brown et al. [12] showed how it would be possible for spammers to craft targeted spam by leveraging the information available in online social networks. As for Twitter, Krishnamurthy et al. studied the network, providing some characterization of Twitter users [14]. Yardi et al. [18] ran an experiment on Twitter spam. They created a popular hashtag on Twitter, and observed that spammers started using it in their messages. They also discuss some features that might allow one to distinguish a spammer from legitimate users, such as node degree and frequency of messages. Another work that studied social network spam using honey-profiles was conducted by Webb et al. in 2008 [17]. For this experiment, 51 profiles were created on MySpace, which was the largest social network at the time. The study showed a significant spam activity. The honey-profiles were contacted by 1,570 spam bots over a five-month period.

Compared to their work, our study is substantially larger in size and covers three major social networks, and the honeypot population we used is representative of the average population of these networks, both from an age and nationality point of view. Moreover, we leverage our observation to develop a system able to detect spammers on social networks.

This system has detected thousands of spam accounts on Twitter, which have been subsequently deleted.

3. DATA COLLECTION

The first goal of our paper was to understand the extent to which spam is a problem on social networks, as well as the characterization of spam activity. To this end, we created 900 profiles on Facebook, MySpace, and Twitter, 300 on each platform. The purpose of these accounts was to log the traffic (e.g., friend requests, messages, invitations) they receive from other users of the network. Due to the similarity of these profiles to honeypots [4], we call these accounts *honey-profiles*.

3.1 Honey-Profiles

Our goal was to create a number of honey-profiles that reflect a representative selection of the population of the social networks we analyzed. To this end, we first crawled each social network to collect common profile data.

On Facebook, we joined 16 geographic networks, using a small number of manually-created accounts. This was possible because, at the time, geographic networks were still available. Since we wanted to create profiles reflecting a diverse population, we joined networks on all continents (except Antarctica and Australia): the Los Angeles and New York networks for North America, the London, France, Italy, Germany, Russia, and Spain ones for Europe, the China, Japan, India, and Saudi Arabia ones for Asia, the Algeria and Nigeria ones for Africa, and the Brazil and Argentina networks for South America. For each network, we crawled 2,000 accounts at random, logging names, ages, and gender (which is the basic information required to create a profile on Facebook). Afterwards, we randomly mixed this data (names, surnames, and ages) and created the honey-profiles. Gender was determined by the first name. Each profile was assigned to a network. Accounts created using data from a certain network were assigned either to this network or to a network where the main language spoken was the same (e.g., profiles created from accounts in the France network were used in networks associated with francophone countries). This was a manual process. For larger networks (e.g., New York, Germany, Italy) up to three accounts were created, while only one account was set up for smaller ones. In total, we created 300 accounts on the Facebook platform.

On MySpace, we crawled 4,000 accounts in total. This was easier than on Facebook because, as mentioned in Section 2.2, most profile pages are public. Similar to Facebook, our aim was to generate "average" profiles based on the user population of the social network. After data collection, we looked for common names and ages from profiles with different languages, and created profiles in most nations of the world. We created 300 accounts on MySpace for our experiment.

While on Facebook and MySpace, birth date and gender are needed for registration, on Twitter, the only information required for signing up is a full name and a profile name. Therefore, we did not find it necessary to crawl the social network for "average" profile information, and we simply used first names and surnames from the other social networks. For each account, the profile name has been chosen as a concatenation of the first and last name, plus a random number to avoid conflicts with already existing accounts. Similarly to the other networks, we created 300 profiles.

We did not create more than 300 profiles on each network because registration is a semi-automated process. More precisely, even though we could automatically fill the forms required for registration, we still needed a human to solve the CAPTCHAs involved in the process.

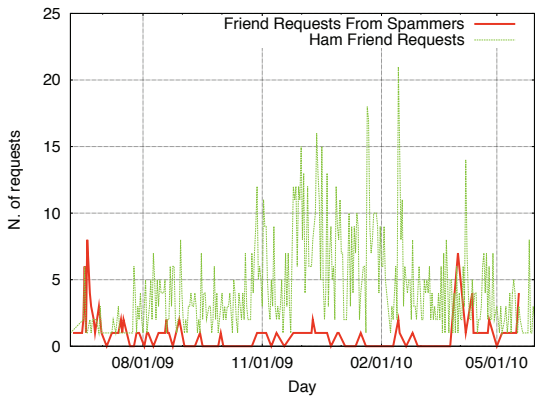
3.2 Collection of Data

After having created our honey-profiles, we ran scripts that periodically connected to those accounts and checked for activity. We decided that our accounts should act in a passive way. Therefore, we did not send any friend requests, but accepted all those that were received.

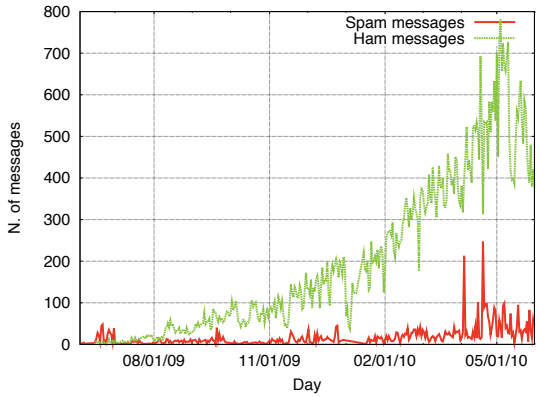
In a social network, the first action a malicious user would likely execute to get in touch with his victims is to send them a friend request. This might be done to attract the user to the spammer's profile to view the spam messages (on MySpace) or to invite her to accept the friendship and start seeing the spammer's messages in her own feed (on Facebook and Twitter).

After having acknowledged a request (i.e., accepted the friendship on Facebook and MySpace or started following the user on Twitter), we logged all the information needed to detect malicious activity. More precisely, we logged every email notification received from the social networks, as well as all the requests and messages seen on the honey-profiles. On some networks, such as Facebook, the notifications and messages might be of different types (e.g., application and group invitations, video posts, status messages, private messages), while on other platforms, they are more uniform (e.g., on Twitter, they are always short text messages). We logged all types of requests on Facebook, as well as wall posts, status updates, and private messages. On MySpace, we recorded mood updates, wall posts, and messages. On Twitter, we logged tweets and direct messages.

Our scripts ran continuously for 12 months for Facebook (from June 6, 2009 to June 6, 2010), and for 11 months for MySpace and Twitter (from June 24, 2009 to June 6, 2010), periodically visiting each account. The visits had



(a) Friend requests received.



(b) Messages received.

Figure 1: Activity observed on Facebook

to be performed slowly (approximately one account visited every 2 minutes) to avoid being detected as a bot by the social networking site and, therefore, having the accounts deleted.

4. ANALYSIS OF COLLECTED DATA

As mentioned previously, the first action that a spammer would likely execute is to send friend requests to her victims. Only a fraction of the contacted users will acknowledge a request, since they do not know the real-life person associated with the account used by the bot[1]. On Twitter, the concept of friendship is slightly different, but the modus operandi of the spammers is the same: they start following victims, hoping that they will follow them back, starting to receive the spam content. From the perspective of our analysis, friendships and mutual follow relationships are equivalent. When a user accepts one of the friend requests, she lets the spammer enter her network of trust. In practice, this action has a major consequence: The victim starts to see messages received from the spammer in her own news/message feed. This kind of spamming is very effective, because the spammer has only to write a single message (e.g., a status update on Facebook), and the message appears in the feeds of all the victims. Depending on the social network, the nature of these messages can change: they are *status updates* on Facebook, *status* or *mood updates* on MySpace, and *tweets* on Twitter.

[1]We assume that most spam accounts are managed in an automated fashion. Therefore, from this point on, we will use the terms spam profile and *bots* interchangeably.

Network	Overall	Spammers
Facebook	3,831	173
MySpace	22	8
Twitter	397	361

Table 1: Friend requests received on the various social networks.

Network	Overall	Spammers
Facebook	72,431	3,882
MySpace	25	0
Twitter	13,113	11,338

Table 2: Messages received on the various social networks.

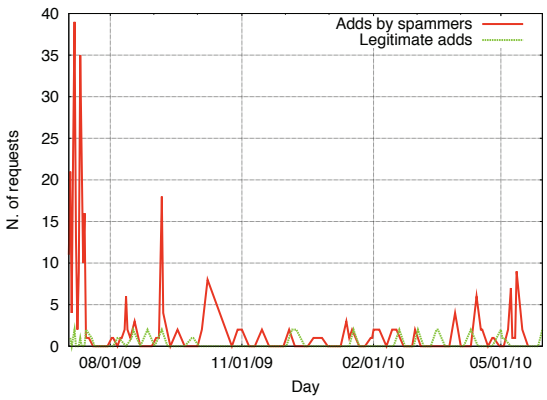
During our study, we received a total of 4,250 friend requests. As can be seen in Table 1, the amount of requests varies from network to network. This might be caused by the different characteristics of the various social networks. As one would expect, we observed the largest amount of requests on Facebook, since it has the largest user base. Surprisingly, however, the majority of these requests proved not to come from spam bots, but from real users, looking for popularity or for real persons with the same name as one of our honey-profiles. Another surprising finding is that, on MySpace, we received a very low number of friend requests. It is not clear what is the reason of the disparity between this social network and Facebook, since MySpace also provides a mechanism to easily post messages on users' pages. Daily statistics for friend requests received on Facebook and Twitter are shown in Figures 1(a) and 2(a).

Information about the logged messages is shown in Table 2. Overall, we observed 85,569 messages. Again, there is a big disparity between the three social networks. On Twitter, interestingly, we recorded the largest amount of spam messages. Given the smaller size of the network's user base, this is surprising. Daily statistics for messages received on Facebook are shown in Figure 1(b), while those for Twitter are reported in Figure 2(b). We do not show a graph for MySpace because the number of messages we received was very low.

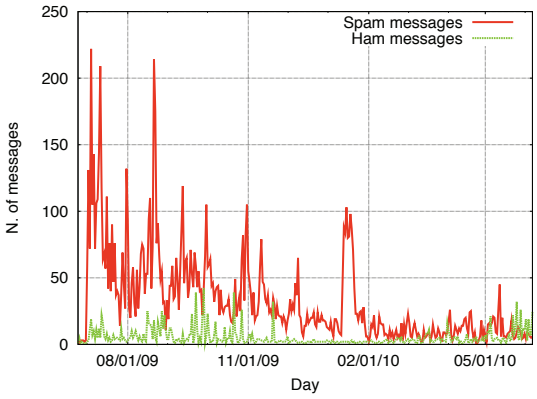
On Facebook, we also observed a fair amount of invitations to applications, groups, and events, as well as posting of photos and videos in our honey-profiles' feeds. However, since none of them were spam, we ignored them for the subsequent analysis.

4.1 Identification of Spam Accounts

Tables 1 and 2 show the breakdown of requests that were received by our honey-profiles. We can see that the honey-profiles did not only receive friend requests and messages from spammers, but also a surprising amount from legitimate accounts. Even if friend requests are unsolicited, they are not always the result of spammers who reach out. In particular, many social network users aim to increase their popularity by adding as friends people they do not know. On



(a) Users starting following honey-profiles



(b) Messages received

Figure 2: Activity observed on Twitter.

Facebook, since all our honey-profiles were members of a geographic network (as long as these were available), it is also possible that people looking for local "friends" would have contacted some of our accounts. In particular, we observed that this occurs with more frequency on smaller networks (in particular, some Middle Eastern and African ones). Moreover, since we picked random combinations of first and last names, it happened that some of our honey-profiles had the same name as a real person, and, as a consequence, the account was contacted by real friends of this person. Since not all friend requests and messages are malicious, we had to distinguish between spammers and benign users.

To discriminate between real users and spam bots, we started to manually check all the profiles that contacted us. During this process, we noticed that spam bots share some common traits, and formalized them in features that we then used for automated spam detection. We will describe these features in detail in Section 5.

We found that, of the original 3,831 accounts that contacted us on Facebook, 173 were spammers. Moreover, on Facebook, during the last months of logging, the ratio of spam messages compared to legitimate ones dramatically dropped. The reason is that when a legitimate user adds our honey-profile to her friend list, this honey-profile starts appearing on her friends' pages as a friend suggestion. This leads to a number of additional friend requests (and messages) from real users. On MySpace, we detected 8 spammers. On Twitter, we detected 361 spammers out of 397 contacts.

4.2 Spam Bot Analysis

The spam bots that we identified showed different levels of activity and different strategies to deliver spam. Based on their spam strategy, we distinguish four categories of bots:

1. **Displayer**: Bots that do not post spam messages, but only display some spam content on their own profile pages. In order to view spam content, a victim has to manually visit the profile page of the bot. This kind of bots is likely to be the least effective in terms of people reached. All the detected MySpace bots belonged to this category, as well as two Facebook bots.

2. **Bragger**: Bots that post messages to their own feed. These messages vary according to the networks: on Facebook, these messages are usually status updates, while on Twitter these are the tweets. The result of this action is that the spam message is distributed and shown on all the victims' feeds. However, the spam is not shown on the victim's profile when the page is visited by someone else (i.e., a victim's friends). Therefore, the spam campaign reaches only victims who are directly connected with the spam bot. 163 bots on Facebook belonged to this category, as well as 341 bots on Twitter.

3. **Poster**: Bots that send a direct message to each victim. This can be achieved in different ways, depending on the social network. On Facebook, for example, the message might be a post on a victim's wall. The spam is shown on the victims feed, but, unlike the case of a "bragger", can be viewed also by victim's friends visiting her profile page. This is the most effective way of spamming, because it reaches a greater number of users compared to the previous two. Eight bots from this category have been detected, all of them on the Facebook network. Koobface-related messages also belong to this category (see [9]).

4. **Whisperer**: Bots that send private messages to their victims. As for "poster" bots, these messages have to be addressed to a specific user. The difference, however, is that this time the victim is the only one seeing the spam message. This type of bots is fairly common on Twitter, where spam bots send direct messages to their victim. We observed 20 bots of this kind on this network, but none on Facebook and MySpace.

We then examined the activity of spam bots on different networks. On Facebook, we observed an average of 11 spam messages per day, while, on Twitter, the average number of messages observed was 34. On MySpace, we did not observe any direct spam message. The reason is that all the spam bots on MySpace are "displayers." The difference between Twitter and Facebook activity is caused by the apparently different responses of the two social networks to spam. More precisely, we observed that Facebook seems to be much more aggressive in fighting spam. This is demonstrated by the fact that, on Facebook, the average lifetime of a spam account was four days, while on Twitter, it was 31 days. On

MySpace, no spam accounts have been deleted during our observation.

As shown in Figures 1(a) and 2(a), many spam requests arrived during the first days of our experiment, especially on Facebook. All the early-days spammers have been quickly deleted from Facebook (the one with the longest life lasted one month), while most of the Twitter ones were deleted only after we flagged them to their spam team.

It is also interesting to look at the time of the day when messages and friend requests are sent. The reason is that bots might get activated periodically or at specific times to send their messages. Benign activity, on the other hand, follows the natural diurnal pattern. During our observation, we noticed that some bots showed a higher activity around midnight (GMT -7), while in the same period of time, the ham messages registered a low.

Another way to study the effectiveness of spam activity is to look at how many users acknowledged friend requests on the different networks. On Facebook, the average number of confirmed friends of spam bots is 21, on MySpace it is 31, while on Twitter, it is 350. We assume that the difference in number of people reached is probably due to the different lifetime of the bots in the different networks. The low activity of the bots on MySpace might be the cause of both the low numbers of bots detected on that network and their longer lifetime.

We identified two kinds of bot behavior: stealthy and greedy bots. Greedy ones include a spam content in every message they send. They are easier to detect, and might lead users to flag bots as spammers or to revoke their friendship status. Stealthy bots, on the other hand, send messages that look legitimate, and only once in a while inject a malicious message. Since they look like legitimate profiles, they might convince more people to accept and maintain friendships.

Of the 534 spam bots detected, 416 were greedy and 98 were stealthy (note that ten spam profiles were "displayers," and 20 were "whisperers." These bots, therefore, did not use updates or tweets to spam).

Another interesting observation is that spam bots are usually less active than legitimate users. This probably happens because sending out too many messages would make detection by the social network too easy. For this reason, most spam profiles we observed, both on Facebook and Twitter, sent less than 20 messages during their life span.

While observing Facebook spammers, we also noticed that many of them did not seem to pick victims randomly, but, instead, they seemed to follow certain criteria. In particular, most of their victims happened to be male. This was particularly true for campaigns advertising adult websites. Since Facebook does not provide an easy way to search for people based on gender, the only way spammers can identify their victims is by looking for male first names. This intuition led us to another observation. The list of victims targeted by these bots usually shows an anomalous repetition of people with the same first name (e.g., tens of profiles with only four different given names). This might happen because spam bots are given lists of first names to target. In addition, Facebook people search does not make a difference between first and last name while searching. For this reason, these gender-aware bots sometimes targeted female users who happened to have a male name as last name (e.g., Wayne).

Mobile Interface.

Most social networking sites have introduced techniques to prevent automatic account generation and message sending. On Facebook, for example, a user is required to solve a CAPTCHA [5] every time she tries to send a friend request. A CAPTCHA has to be solved also every time an account is created. Moreover, the site uses a very complicated JavaScript environment that makes it difficult for bots to interact with the pages. On the other hand, the complexity of these sites made them not very attractive to mobile Internet users, who use less powerful devices and slower connections.

To attract more users and to make their platform more accessible from any kind of device, major social networks launched mobile versions of their sites. These versions offer the main functionality of the complete social networking sites, but in a simpler fashion. To improve usability, no JavaScript is present on these pages, and no CAPTCHAs are required to send friend requests. This has made social networks more accessible from everywhere. However, the mobile environment provides spammers with an easy way to interact with these sites and carry out their tasks. This is confirmed by our analysis: 80% of bots we detected on Facebook used the mobile site to send their spam messages. However, to create an account, it is still necessary to go through the non-mobile version of the site. For Twitter spam, there is no need for the bots to use the mobile site, since an API to interact with the network is provided, and, in any case, there is no need to solve CAPTCHAs other than the one needed to create a profile.

5. SPAM PROFILE DETECTION

Based on our understanding of spam activity in social networks, the next goal was to leverage these insights to develop techniques to detect spammers in the wild. We decided to focus on detecting "bragger" and "poster" spammers, since they do not require real profiles for detection, but are just detectable by looking at their feeds. We used machine learning techniques to classify spammers and legitimate users. To detect whether a given profile belongs to a spammer or not, we developed six features, which are:

FF ratio (R): The first feature compares the number of friend requests that a user sent to the number of friends she has. Since a bot is not a real person, and, therefore, nobody knows him/her in real life, only a fraction of the profiles contacted would acknowledge a friend request. Thus, one would expect a distinct difference between the number of friend requests sent and the number of those that are acknowledged. More precisely, we expect the ratio of friend requests to actual friends to be large for spammers and low for regular users. Unfortunately, the number of friend requests sent is not public on Facebook and on MySpace. On Twitter, on the other hand, the number of users a profile started to follow is public. Therefore, we can compute the ratio $R = \textit{following} \ / \ \textit{followers}$ (where following, in the Twitter jargon, is the number of friend requests sent, and followers is the number of users who accepted the request).

URL ratio (U): The second feature to detect a bot is the presence of URLs in the logged messages. To attract users to spam web pages, bots are likely to send URLs in their messages. Therefore, we introduce the ratio U as:

$$U = \textit{messages\_containing\_urls} \ / \ \textit{total\_messages}.$$

Since, in the case of Facebook, most messages with URLs (link and video share, group invitations) contain a URL to other Facebook pages, we only count URLs pointing to a third party site when computing this feature.

Message Similarity (S): The third feature consists in leveraging the similarity among the messages sent by a user. Most bots we observed sent very similar messages, considering both message size and content, as well as the advertised sites. Of course, on Twitter, where the maximum size of the messages is 140 characters, message similarity is less significant than on Facebook and MySpace, where we logged messages up to 1,100 characters. We introduced the similarity parameter S, which is defined as follows:

$$S = \frac{\sum_{p \in P} c(p)}{l_a l_p},$$

where P is the set of possible message-to-message combinations among any two messages logged for a certain account, p is a single pair, $c(p)$ is a function calculating the number of words two messages share, l_a is the average length of messages posted by that user, and l_p is the number of message combinations. The idea behind this formula is that a profile sending similar messages will have a low value of S.

Friend Choice (F): The fourth feature attempts to detect whether a profile likely used a list of names to pick its friends or not. We call this feature F, and we define it as:

$$F = \frac{T_n}{D_n},$$

where T_n is the total number of names among the profiles' friend, and D_n is the number of distinct first names. Our observation showed that legitimate profiles have values of this feature that are close to 1, while spammers might reach values of 2 or more.

Messages Sent (M): We use the number of messages sent by a profile as a feature. This is based on the observation that profiles that send out hundreds of messages are less likely to be spammers, given that, in our initial analysis, most spam bots sent less that 20 messages.

Friend Number (FN): Finally we look at the number of friends a profile has. The idea is that profiles with thousands of friends are less likely to be spammers that the ones with a few.

Given our general set of features, we built two systems to detect spam bots on Facebook and Twitter. Since there are differences between these two social networks, some features had to be slightly modified to fit the characteristics of the particular social network. However, the general approach remains the same. We used the Weka framework [7] with a Random Forest algorithm [11] for our classifier. We chose this algorithm because it was the one that gave the best accuracy and lowest false positive ratio when we performed the cross-validation of the training set.

5.1 Spam Detection on Facebook

The main issue when analyzing Facebook is to obtain a suitable amount of data to analyze. Most profiles are private, and only their friends can see their walls. At the beginning of this study, geographic networks were still available, but they were discontinued in October 2009. Therefore, we used data from various geographic networks, crawled between April 28 and July 8 2009, to test our approach.

Since on Facebook the number of friend requests sent out is not public, we could not apply the R feature.

We trained our classifier using 1,000 profiles. We used the 173 spam bots that contacted our honey-profiles as samples for spammers, and 827 manually checked profiles from the Los Angeles network as samples for legitimate users. A 10-fold cross validation on this training data set yielded an estimated false positive ratio of 2% and a false negative ratio of 1%. We then applied our classifier to 790,951 profiles, belonging to the Los Angeles and New York networks. We detected 130 spammers in this dataset. Among these, 7 were false positives. The reason for this low number of detected spammers might be that spam bots typically do not join geographic networks. This hypothesis is corroborated by the fact that among the spam profiles that contacted out honey profiles, none was a member of a geographic network. We then randomly picked 100 profiles, classified as legitimate. We manually looked at them to search for false negatives. None of them turned out to be a spammer.

5.2 Spam Detection on Twitter

On Twitter, is much easier to obtain data than on Facebook, since most profiles are public. This gave us the possibility to develop a system that is able to detect spammers in the wild. The results of our analysis were then sent to Twitter, who verified that the accounts were indeed sending spam and removed them.

To train our classifier, we picked 500 spam profiles, coming either from the ones that contacted our honey profiles, or manually selected from the public timeline. We included profiles from the public timeline to increase diversity among spam profiles in our training dataset. Among the profiles from the public timeline, we chose the ones that stood out from the average for at least one of the R, U, and S features. We also picked 500 legitimate profiles from the public timeline. This was a manual process, to make sure that no spammers were miscategorized in the training set. The R feature was modified to reflect the number of followers a profile has. This was done because legitimate profiles with a fairly high number of followers (e.g., 300), but following thousands of other profiles, have a high value of R. This is a typical situation for legitimate accounts following news profiles, and would have led to false positives in our system. Therefore, we defined a new feature R', which is the R value divided by the number of followers a profile has. We used it instead of R for our classification.

After having trained the classifier, it was clear that the F feature is not useful to detect spammers on Twitter, since both spammers and legitimate profiles in the training set had very similar values for this parameter. This suggests that Twitter spam bots do not pick their victims based on their name. Therefore, we removed the F feature from the Twitter spam classifier. A 10-fold cross validation for the classifier with the updated feature set yielded an estimated false positive ratio of 2.5% and a false negative ratio of 3% on the training set.

Given the promising results on the training set and the possibility to access most profiles, we decided to use our classifier to detect spammers in real time on Twitter. The main problem we faced while building our system was the crawling speed. Twitter limited our machine to execute only 20,000 API calls per hour. Thus, to avoid wasting our limited API calls, we executed Google searches for the most common words in tweets sent by the already detected spammers, and we crawled those profiles that were returned as

results. This approach has the problem that we can only detect profiles that send tweets similar to those of previously observed bots. To address this limitation, we created a public service where Twitter users can flag profiles as spammers. After a user has flagged someone as a spammer, we run our classifier on this profile data. If the profile is detected as a spammer, we add this profile to our detected spam set, enabling our system to find other profiles that sent out similar tweets.

Every time we detected a spam profile, we submitted it to Twitter. During a period of three months, from March 06, 2010 to June 06, 2010, we crawled 135,834 profiles, detecting 15,932 of those as spammers. We sent this list of profiles to Twitter, and only 75 were reported by them to be false positives. All the other submitted profiles were deleted. In order to evaluate the false negative ratio, we randomly picked 100 profiles, classified as legitimate by our system. We then manually checked at them, finding out that 6 were false negatives.

To show that our targeted crawling does not affect our accuracy or false positive ratio, but just narrowed down the set of profiles to crawl, we picked 40,000 profiles at random from the public timeline and crawled them. Among these, we detected 102 spammers, with a single false positive. We can see that our crawling is effective, since the percentage of spammers in our targeted (crawled) dataset is 11%, whereas in the random set, it is 0.25%. On the other hand, the false positive ratio on in both datasets is similarly low.

5.3 Identification of Spam Campaigns

After having identified single spammers, we analyzed the data to identify larger-scale spam campaigns. With "spam campaign," we refer to multiple spam profiles that act under the coordination of a single spammer. We consider two

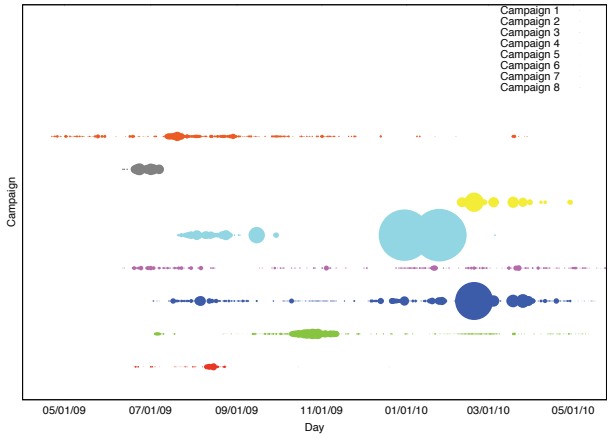


Figure 3: Activity of campaigns over time.

bots posting messages with URLs pointing to the same site as being part of the same campaign. Most bots hide the real URL that their links are pointing to by using URL shortening services (for example, *tinyurl* [6]). This is typically done to avoid easy detection by social networks administrators and by the users, as well as to meet the message length requirements of some platforms (in particular, Twitter). To determine the actual site that a shortened URL points to, we visited all the URLs that we observed. Then, we clustered all the profiles that advertised the same page. We list the top eight campaigns, based on the number of observed messages, in Table 3. Since we had most detections on Twitter, these campaigns targeted that network. It is interesting to notice, however, that bots belonging to three of them were observed on Facebook as well.

Some campaigns showed a large number of bots, each sending a few messages per day, while others send many messages using few bots. In addition, the fact that bots of a campaign can act in a stealthy or greedy way (see Section 4.2) leads to significantly different outcomes. Greedy bots that send spam with each message are easier to detect by the social network administrators. On the other hand, a low-traffic spam campaign is not easy to detect. For example, the bots from Campaign 1 sent 0.79 messages per day, while the bots from the second campaign sent 0.08 messages per day on average. The result was that the bots from Campaign 1 have an average lifetime of 25 days, while the bots of Campaign 2 lasted 135 days on average. In addition, Campaign 2 reached more victims, as shown by an average of 94 friends (victims) per bot, while Campaign 1 only reached 52. This suggests that a relationship exists between the lifetime of bots and the number of victims targeted. Clearly, an effective campaign should be able to reach many users, and having bots that live longer might be a good way to achieve this objective.

From the point of view of victims reached, stealthy campaigns are more effective. Campaigns 4 and 7 both used a stealthy approach. Of the messages sent, only 20-40% contained spam content. As a result, bots from Campaign 4 had an average lifetime of 120 days, and started following 460 profiles each. Among these, 87 users on average followed the bots back. Campaign 7 was the most effective among Twitter campaigns, both considering the number of victims and the average bot lifetime. To achieve this, this campaign combined a low rate of messages per day with a stealth way of operating. The bots in this campaign have an average lifetime of 198 days and 1,787 victims, of which, on average, 112 acknowledged the friend request.

From the observations of the various campaigns, we developed a metric that allows us to predict the success of a campaign. We consider a campaign successful if the bots belonging to it have a long lifetime. For this metric, we introduce the parameter G_c, defined as follows:

$$G_c = \frac{M_d^{-1} \cdot S_d}{((\sqrt{M_d^{-1} \cdot S_d}) + 1)^2}, \; 0 \leq G_c \leq 1.$$

In the above formula, M_d is the average number of messages per day sent and S_d is the ratio of actual spam messages ($0 \leq S_d \leq 1$). Empirically, we see that campaigns with a value of G_c close to 1 have a long lifetime (for example, Campaign 7 has $G_c = 0.88$, while Campaign 2 has $G_c = 0.60$), while for campaigns with a lower value of this parameter, the average lifetime decreases significantly (Campaign 1 has $G_c = 0.28$ and Campaign 5 has $G_c = 0.16$). Thus, we can infer that a value of 0.5 or higher for G_c indicates that a campaign has a good chance to be successful. Of course, if a campaign is active for some time, a social network might develop other means to detect spam bots belonging to it (e.g., a blacklist of the URLs included in the messages).

Activity of bots from different campaigns is shown in Figure 3. Each row represents a campaign. For each day in

#	SN	Bots	# Mes.	Mes./day	Avg. vic.	Avg. lif.	G_c	Site adv.
1	T	485	1,020	0.79	52	25	0.28	Adult Dating
2	T	282	9,343	0.08	94	135	0.60	Ad Network
3	T,F	2,430	28,607	0.32	36	52	0.42	Adult Dating
4	T	137	3,213	0.15	87	120	0.56	Making Money
5	T,F	5,530	83,550	1.88	18	8	0.16	Adult Site
6	T,F	687	7,298	1.67	23	10	0.18	Adult Dating
7	T	860	4,929	0.05	112	198	0.88	Making Money
8	T	103	5,448	0.4	43	33	0.37	Ad Network

Table 3: Spam campaigns observed.

which we observed some activity from that campaign, a circle is drawn. The size of circles varies according to the number of messages observed that day. As can be seen, some campaigns have been active over the entire period of the study, while some have not been so successful.

We then tried to understand how bots choose their victims. The behavior seems not to be uniform for the various campaigns. For example, we noticed that many victims of Campaign 2 shared the same hashtag (e.g., "#iloveitwhen") in their tweets. Bots might have been crawling for people sending messages with such tag, and started following them. On the other hand, we noticed that Campaigns 4 and 5 targeted an anomalous number of private profiles. Looking at their victims, 12% of them had a private profile, while for a random picked set of 1,000 users from the public timeline, this ratio was 4%. This suggests that bots from these campaigns did not crawl any timeline, since tweets from users with a private profile do not appear on them.

6. CONCLUSIONS

Social networking sites have millions of users from all over the world. The ease of reaching these users, as well as the possibility to take advantage of the information stored in their profiles, attracts spammers and other malicious users.

In this paper, we showed that spam on social networks is a problem. For our study, we created a population of 900 honey-profiles on three major social networks and observed the traffic they received. We then developed techniques to identify single spam bots, as well as large-scale campaigns. We also showed how our techniques help to detect spam profiles even when they do not contact a honey-profile. We believe that these techniques can help social networks to improve their security and detect malicious users. In fact, we develop a tool to detect spammers on Twitter. Providing Twitter the results of our analysis thousands of spamming accounts were shut down.

7. ACKNOWLEDGMENTS

This work was supported by the ONR under grant N000140911042 and by the National Science Foundation (NSF) under grants CNS-0845559 and CNS-0905537.

8. REFERENCES

[1] Alexa top 500 global sites. `http://www.alexa.com/topsites`.

[2] Compete site comparison. `http://siteanalytics.compete.com/facebook.com+myspace.com+twitter.com/`.

[3] Facebook statistics. `http://www.facebook.com/press/info.php?statistics`.

[4] Honeypots. `http://en.wikipedia.org/wiki/Honeypot\_computing`.

[5] The recaptcha project. `http://recaptcha.net/`.

[6] Tinyurl. `http://tinyurl.com/`.

[7] Weka - data mining open source program. `http://www.cs.waikato.ac.nz/ml/weka/`.

[8] Sophos facebook id probe. `http://www.sophos.com/pressoffice/news/articles/2007/08/facebook.html`, 2008.

[9] J. Baltazar, J. Costoya, and R. Flores. Koobface: The largest web 2.0 botnet explained. 2009.

[10] L. Bilge, T. Strufe, D. Balzarotti, and E. Kirda. All your contacts are belong to us: Automated identity theft attacks on social networks. In *World Wide Web Conference*, 2009.

[11] L. Breiman. Random forests. In *Machine Learning*, 2001.

[12] G. Brown, T. Howe, M. Ihbe, A. Prakash, and K. Borders. Social networks and context-aware spam. In *ACM Conference on Supportive Cooperative Work*, 2008.

[13] T.N. Jagatic, N.A. Johnson, M. Jakobsson, and T.N. Jagatif. Social phishing. *Comm. ACM*, 50(10):94–100, 2007.

[14] B. Krishnamurthy, P. Gill, , and M. Aritt. A few chirps about twitter. In *USENIX Workshop on Online Social Networks*, 2008.

[15] S. Moyer and N. Hamiel. Satan is on my friends list: Attacking social networks. `http://www.blackhat.com/html/bh-usa-08/bh-usa-08-archive.html`, 2008.

[16] Harris Interactive Public Relations Research. A study of social networks scams. 2008.

[17] S. Webb, J. Caverlee, , and C.Pu. Social honeypots: Making friends with a spammer near you. In *Conference on Email and Anti-Spam (CEAS 2008)*, 2008.

[18] S. Yardi, D. Romero, G. Schoenebeck, and D. Boyd. Detecting spam in a twitter network. *First Monday*, 15(1), 2010.

Toward Worm Detection in Online Social Networks

Wei Xu
Pennsylvania State University
University Park, Pennsylvania
wxx104@cse.psu.edu

Fangfang Zhang
Pennsylvania State University
University Park, Pennsylvania
fuz104@cse.psu.edu

Sencun Zhu
Pennsylvania State University
University Park, Pennsylvania
szhu@cse.psu.edu

ABSTRACT

Worms propagating in online social networking (OSN) websites have become a major security threat to both the websites and their users in recent years. Since these worms exhibit unique propagation vectors, existing Internet worm detection mechanisms cannot be applied to them. In this work, we propose an early warning OSN worms detection system, which leverages both the propagation characteristics of these worms and the topological properties of online social networks. Our system can effectively monitor the entire social graph by keeping only a small number of user accounts under surveillance. Moreover, the system applies a two-level correlation scheme to reduce the noise from normal user communications such that infected user accounts can be identified with a higher accuracy. Our evaluation on the real social graph data obtained from Flickr indicates that by monitoring five hundreds users out of 1.8 million users, the proposed detection system can detect the burst of an OSN worm when less than 0.13% of total user accounts are infected. Besides, by adopting simple countermeasures, the detection system is also shown to be very helpful for worm containment.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Security

Keywords

Worm Detection, Online Social Networks, Decoy, Dominating Set, Early Warning

1. INTRODUCTION

The popularity of online social networking (OSN) websites has been boosted in recent years. For example, Facebook has more than 400 million active users [4] worldwide, MySpace has more than 70 million members [9] in U.S., Twitter has more than 75 million users [13] and LinkedIn has more than 44 million members [7]. This rapid growth of OSN websites has given rise to a number of security attacks. A representative example is a worm named Koobface, which spread in Facebook and MySpace in August 2008 [6] [10] [14]. It has generated 56 variants and has infected many other websites such as Tagged, Friendster, MyYearBook, Fubar.com, Hi5 and Bebo [1] [3] [8]. Besides Koobface, there are other worms, such as Mikeyy worm [12] and Samy worm [24] that have also caused havoc in OSN websites.

OSN websites have become an attractive target for these worms (hereinafter referred to as *OSN worms*) because of the following properties of the online social networks. First, online social networks are small-world networks [21], which means they have the properties of small average shortest path length and high clustering. The small average shortest path length property can reduce the propagation time from one user account to another user account. Meanwhile, the high clustering property suggests that users are tightly connected together, which facilitates the explosion of OSN worms. Second, online social networks are also scale-free networks [25], which are a class of power-law networks where high-degree nodes tend to connect with other high-degree nodes. When an OSN worm infects the account of a popular user (i.e., user with a large number of friends), this scale-free property suggests that the worm can infect another popular account shortly. As a result, the worm can achieve exponential growth by propagating to the large friends set of these popular users. Moreover, OSN worms also leverage social engineering [6] to increase the authenticity of worm messages.

By exploiting these properties of online social networks, OSN worms exhibit fast spreading characteristics similar to the ones observed in Internet worms. However, existing Internet worm detection mechanisms can not be directly applied to detecting OSN worms. This is because Internet worm detection heavily relies on the unique patterns of worm scanning traffic or the misbehavior of infected hosts, but neither of them can be observed in the propagation of an OSN worm. From the perspective of OSN websites (i.e., the server side), an infected user account does nothing but sending messages or posting updates as normal users do when the actual infection is taking place in the browser (i.e., the client side). This makes the detection of OSN worms a new and interesting problem.

In this paper, we propose an early warning OSN worm

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

detection system. Early warning is essential for OSN worm detection because it provides administrators the opportunity to apply worm containment and elimination measures without affecting a large portion of user accounts. Meanwhile, achieving early warning in the detection of fast spreading worms is also a challenging problem [20].

Our approach leverages the properties of online social networks and the inherent propagation characteristics of OSN worms. More specifically, we first build a surveillance network based on the small-world and scale-free properties of the social graph to collect suspicious worm propagation evidence. We maximize the surveillance coverage by monitoring only a small fraction of user accounts. These accounts are selected by investigating the vertex properties of the social graph. We also realize that detection based on suspicious evidence alone is prone to high false positives, because worm activities are highly likely to be drown out in normal user activities. As such, we further propose a scheme to effectively filter out the noise in the surveillance network. The implementation of the system is flexible, it can either reside on a dedicated server or be distributed within the OSN websites.

Our experimental results based on a real-life social graph dataset consisting of over 1.8 million users and 22.6 million friend links demonstrated the effectiveness of our detection system. The outbreak of an OSN worm can be detected when only 0.13% of the total user accounts are infected. Moreover, our detection system is simple in design, like a lightweight network intrusion detection system (NIDS), and it only needs to process a relatively small number of (suspicious) propagation evidence collected from the users in an EDS. We believe this could be a practical and easy solution for OSN websites that are currently struggled with OSN worms.

The rest of the paper is organized as follows: Section 2 provides the background on various OSN worms and characterizes their propagation vectors. We describe the design of the proposed detection system in Section 3. In Section 4, we evaluate its effectiveness in worm detection. We discuss several limitations of our work in Section 5 and describe some related works in Section 6. Finally, we conclude our work in Section 7.

2. BACKGROUND

Various OSN worms spread themselves by exploiting different features of the OSN websites. Nonetheless, their propagation vectors share certain similarities, which will be characterized by examining two representative OSN worms: Koobface and Mikeyy.

Figure 1 illustrates the propagation flow of Koobface in Facebook. User A receives a worm message from one of her friends (step 1) after this friend was infected by Koobface. Within this worm message, there is a link to a video clip hosted on a fake YouTube website. When user A clicks that link, the browser is redirected to the fake YouTube webpage (step 2), where the user is prompted by a request to install an update for "Adobe Flash player" plugin, which is actually a malware. After user A installs the claimed browser plugin (step 3), Koobface infects user A's Facebook account and iterate its infection cycle by sending similar worm messages to all the friends in user A's profile (step 4). Actually, besides sending messages, Koobface also sends invitations or composes posts, both of which contain similar worm content.

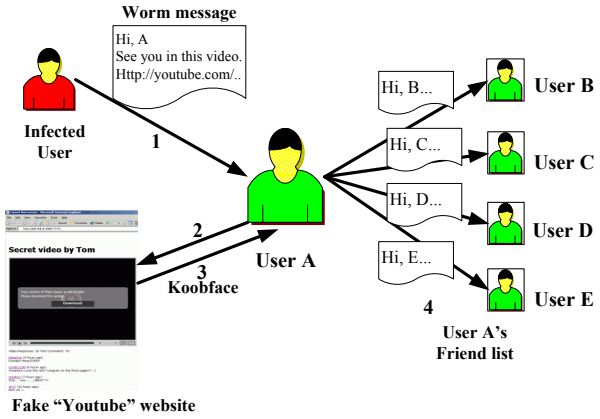


Figure 1: Koobface worm infection cycle

Mikeyy worm propagates by posting updates on an infected user's profile to encourage the "follower" (people who can automatically receive the updates in their profiles) to visit www.StalkDaily.com, which was owned by the attacker. When a follower who is interested in the update clicks to see the poster's profile, a self-replicated JavaScript code is injected into the follower's profile. After the injection, similar updates are posted on the infected follower's profile to repeat the infection cycle.

Despite the differences in the infection vectors of these two worms (E.g., downloading malware versus self-replicating JS code), both of them propagate following the social connections (E.g., friends or followers) of an infected account; in other words, their propagations follow the topology of the online social network (i.e., the social graph). One reason of this similarity is that social connections provide worms an opportunity to exploit social engineering such that the click-through rate of the malicious content can be increased. Besides, as mentioned before, topological properties of online social networks (e.g., small-world) can facilitate the spread of worms. Another similarity shared by both worms is the generation of passively noticeable activities such as worm messages and worm updates. This is because OSN worms are normally generated with certain malicious purposes such as advertising malicious websites or distributing malware.

In this paper, we limit our discussion to OSN worms that exhibit the above two properties. That is, we aim to detect worms that *propagate following the social connections* and *generate passively noticeable worm activities*. We acknowledge that not all existing OSN worms exhibit both the above properties and future OSN worms could take different formats. For example, Samy worm [2], a cross-site scripting (XSS) worm on MySpace, is one which does not generate activities passively noticeable by friends, so it is out of the scope of this work. We believe it would be better addressed by solutions focused on XSS vulnerabilities [24]. Indeed, the vulnerability exploited by Samy worm has been fixed and hence Samy worm does not work now, whereas the OSN worms we are addressing here remain a big threat to OSN sites.

3. SYSTEM DESIGN

In this section, we elaborate the design of the OSN worm

detection system, starting with a system overview.

As suggested by the high clustering property, the neighborhoods (i.e., friends) of most user accounts are densely connected. Therefore, for each neighborhood, our system only needs to monitor the "popular" user (i.e., the one with most friends in a neighborhood) to cover the entire neighborhood. Meanwhile, the scale-free property implies that a user with a large friend set tends to be friends with other users with large friend sets. This indicates that not all the popular users need to be monitored. Indeed, our system will only select a few of such users to maintain the surveillance coverage.

3.1 Overview

The general idea of our detection system is to deploy a disguised surveillance network being part of the online social networks to collect worm propagation evidence and to identify worm infections. Figure 2 illustrates the framework of our detection system, which consists of four major components. The *configuration module* retrieves from the administrator of the OSN website the social graph, based on which it determines where to collect evidence. The *evidence collecting module* gathers suspicious worm propagation evidence observed in an OSN website. The *worm detection module* identifies and reports a worm infection based on the input from the evidence collecting module. When an infection is detected, this module passes an alarm together with the infection information to the administrator of the OSN website via the communication module. The *communication module* provides all the necessary communications between an OSN website and the other modules. We will explain the design details of these modules in the following subsections.

One noteworthy property of our design is that each module only represents a combination of certain functionalities, and these functionalities can be implemented either within a dedicated server or in a distributed way. This property extends the flexibility of the system implementation, and it will be further discussed in Section 4.

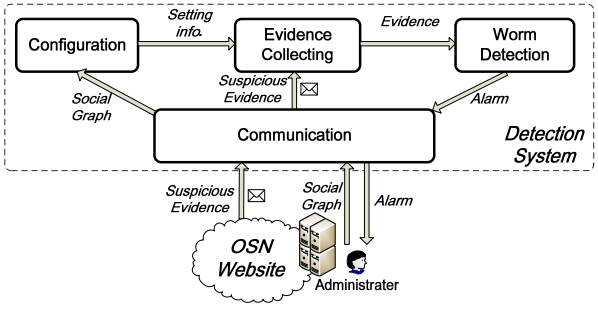


Figure 2: Detection System Overview

3.2 Evidence Collecting Module

The evidence collecting module is in charge of gathering worm propagation evidence (e.g., worm messages, worm updates). However, given the huge amount of information exchanged in an OSN website, the challenge is how to collect only suspicious worm evidence. Since OSN worms follow the social connections in propagation, a friend of an infected user account is more likely to receive worm propagation evidence. To leverage this advantage, we adapt the idea of honeypot here as "decoy friend". A decoy friend is a low-interactive honeypot, and it is created and added into a normal user's friends list by the detection system. When a user account is infected by an OSN worm, decoy friends of that account can receive worm evidence. Similar ideas have been suggested in [30, 16] for other types of networks. In [30], the authors assume decoys only receive malicious messages. However, the same assumption does not hold in our work. In fact, our system treats the collected information from decoys only as suspicious evidence because some normal user activities can also be observed by decoys.

Decoys form a disguised surveillance network. We assign each decoy to be friends with several normal users so that a decoy can not be easily spotted because of its small number of friends. In addition, there are a few practical concerns regarding applying decoy friends in real world OSN websites. The first potential concern is related to user's information privacy because decoys collect suspicious information in the network. However, since users' data are all stored and kept in the OSN websites, we think our system will not cause new data/information leakage. Nevertheless, to alleviate such possible concern, our system will only keep the suspicious information for a short period of time. The second concern is that users might be reluctant to accept decoy friends. As such, a website will need to consult its users before assigning decoy friends to them. In fact, the OSN websites could provide incentives to encourage users to accept decoy friends. After all, both users and the OSN websites try to avoid worm infections for their own benefits. The third concern is on the number of decoy friends to be deployed in an OSN website. Besides user's reluctance, the population of decoys may negatively impact the popularity of an OSN website, because decoy friends do not contribute to any interactive activities such as discussions or communications. To this end, *our system strives to limit the number of decoy friends while preserving the detection effectiveness.* We will discuss this design issue in the next section.

3.3 Configuration Module

The most important function of the configuration module is deploying decoys, which consists of two consecutive steps: selecting normal users and assigning decoy friends to these users. This module also performs other functions such as maintaining the configuration information of the system.

3.3.1 Selecting Normal Users

Because of the practical concerns on applying decoys, our system only selects a small set of users (hereinafter referred to as "selected users set") to be friends with decoys. Meanwhile, the objective of early warning favors a sufficiently large portion of users being kept under the surveillance of our system. Hence, the question is how to choose as small as possible a selected users set to achieve early warning. We formalize this problem in the context of social graph: Given a directed graph $G = (V, E)$, where each vertex denotes a user in the social network and each edge represents a connection between two users [1], choose a minimum set of vertices such that each vertex either belongs to the set or there ex-

[1]in the case of mutual acquaintance between two users, such as in Facebook if A is friends with B, then B is also friends with A, this connection needs to be represented by two directed edges.

ists a path that ends at this vertex and starts from some vertex within the set. The length of this path is at most r hops. This problem is also known as *extended dominating set problem* [29] [19], which is NP-complete.

The choice of r affects the size of the selected users set. Therefore, it is carefully reasoned based on the following study of a real world social graph. Our study first confirms that given the same number of users, a larger r can cover a larger portion of the same social graph, so a larger r is desirable if at all possible. We also find that a worm starts from a single user can infect at most 0.08% of all users in two-hop propagation and 0.26% of all users in three-hop propagation. If our system sets $r = 3$, it is very likely that by the time worm propagation is detected, 0.26% of all users have been infected. This exceeds the early warning criterion (0.19%) suggested in [20], so our system sets $r = 2$ as the coverage radius. Our study also suggests that it is not necessary to cover the entire social graph, because degree distribution of the vertices in a social graph follows power law distribution [25]. This property indicates that many vertices do not even have any connections. For example, over 20% of users in our evaluation data have no connections. These vertices are very unlikely to become the victim of a worm, and the effort of covering such vertices would produce a set with the size comparable to the size of the graph. Based on these studies of the properties of a social graph, we relax the constraint about covering the entire graph and redefine the problem as follow:

Maximum Coverage Problem: Given a social graph $G = (V, E)$ and a number k, choose a set of vertices with size of at most k such that the number of other vertices that are covered by this set with coverage radius $r = 2$ reaches the maximum.

The maximum coverage problem is also NP-complete. The previous extended dominating set problem reduces to it. Since both the scale-free property and the power-law distribution of degree suggest that high-degree vertices are more likely to be infected by a worm than most other low-degree vertices, these high degree vertices should be included in the selected users set with high priority. Besides, research on modelling epidemics in topological networks [15] [26] [31] also indicates that the more edges a node has, with a higher probability it will be infected quickly by an epidemic. These results suggest the following greedy heuristic: At each step, we add one vertex into the set such that the intersection between this vertex's 2-hop coverage and the remaining vertices is maximum. Based on this heuristic, we design the following approximate algorithm.

The time complexity of Algorithm 1 is $O(knm^2)$ where $n = |V|$ and $m = |E|$. In practice, our system pre-processes the social graph to reduce the size of the graph such that the performance of the algorithm could be improved. Besides, since social graphs grow with time, we may run this algorithm periodically(e.g., once a week) to reflect such growth.

Algorithm 1 Maximum Coverage Algorithm

Input: Graph $G = (V, E)$
Output: Monitored user set C

```
C ← ∅
while |C| < k and V ≠ ∅ do
    maxcover ← 0
    for ∀v ∈ V do
        cover_v ← 2-hops coverage of v
        if maxcover < cover_v then
            maxcover ← cover_v
        end if
    end for
    C ← C ⋃ {v}
    V ← V − {v}
    for ∀u ∈ V and (v, u) ∈ E do
        for ∀(u, w) ∈ E do
            V ← V − {w}
        end for
        V ← V − {u}
    end for
    Update degree of each v ∈ V
end while
```

3.3.2 Assigning Decoy Friends

After a candidate selected users set is chosen, the configuration module sends the set to the OSN administrator, who will contact these users (with incentives) and return the final set of users that are willing to accept decoy friends. Upon receiving the final set, this module creates decoy profiles in the OSN website and associates two decoy friends to each user by adding them into the user's friends list (The justification of this scheme is discussed in Section 3.4). This module also modifies the account preference of each decoy friend (according to the setting of the OSN website) so that information received by decoy friends can be collected by the evidence collecting module.

3.4 Worm Detection Module

This module identifies the infected user accounts based on the suspicious worm propagation evidence. To distinguish actual worm evidence from normal user communications, this module applies correlation test on the suspicious evidence. The correlation test is based on similarities in the content and the structure of worm propagation evidence. One reason behind this similarity is that worm messages or updates composed by the same worm usually serve the same purpose (e.g., advertising a malicious link). Another reason is that the automatic message generation algorithms run by worms tend to reuse words and phrases because of the limited size of their candidate words set.

In this module, we employ a two-level spatial-correlation scheme, namely *local correlation* and *network correlation*. To provide necessary information to correlations, our system maintains a data structure called suspicious propagation evidence list (SPEL), which is associated with each selected user. In SPEL, every piece of evidence is stored as a {*decoy friend ID, receiving time, content*} tuple.

Local Correlation: Local correlation performs similarity test among suspicious evidence collected by two decoy friends assigned to the same selected user. The purpose of associating two decoy friends with one user is to offer a local reference such that upon receiving any evidence from one decoy friend, the system can search the other decoy friend's SPEL for similar evidence. One of the following scenarios will happen:

1. Only one of the two decoy friends has received this message. With a high probability, this is worm prop-

agation evidence because a normal user is unlikely to send messages to one of his/her decoy friends, especially given that he/she knows which is a decoy.

2. Both decoy friends have received *similar but not identical* messages. With a high probability, this is worm propagation evidence because only the infected users send customized worm messages to each friend.

3. Both decoy friends receive the same message or update [2]. It could be either a group message or a worm message with the same content. In this case, the scheme resorts to *network correlation* for further identification.

Network Correlation: Network correlation is performed with input from all decoy friends. Upon receiving the same evidence from two decoy friends of a user, network correlation searches for similar evidence by computing the similarity score between the received evidence and any other evidence in the SPELs of other decoys. If similar evidence (e.g., with a similarity over 90%) is found, with high probability, both pieces of evidence can be confirmed as worm propagation evidence. We realize that some normal communications among users may have the similar propagation pattern as worm messages. For example, the outbreak of a large-scale event may cause similar or same messages distributed within an OSN. We will discuss this case in Section 6.

To examine the *similarity* between two pieces of suspicious propagation evidence, our scheme applies a simple measurement of similarity based on the metric of edit distance $editDist()$ [23]. By this measurement, the *similarity* between evidence E_a and evidence E_b can be evaluated as follow:

$$sim(E_a, E_b) = \frac{1}{1 + editDist(E_a, E_b)} \quad (1)$$

where $editDist()$ follows the definition of Levenshtein edit distance [22]. We acknowledge that more complex similarity measurements may generate more accurate results, but since that is not the focus of this paper, we will consider the other metrics in our future work.

Figure 3 gives an example illustrating the mechanism of two-level correlation. In this example, users *A, B, C* belong to the selected users set, and each of them is associated with two decoy friends. We assume that the worm first infects D, then it infects both E and A. After that, B is infected as well. At the time A was infected, if the worm sends customized messages to A's decoy friends *A1* and *A2*, the system can identify A's infection by local correlation between *A1* and *A2* (Scenario 2). If both *A1* and *A2* receive the same message (or update), the system checks the SPELs of all other selected users (e.g., B and C) and compares the received suspicious evidence with stored evidence. Because neither B or C is infected at this time, no match is found. Therefore, the evidence received by both *A1* and *A2* is stored in the SPEL of A. After B is infected, the same procedure is preformed and a match can be found between the evidence stored in the SPEL of A and the evidence just received by *B1* and *B2*. At this time, the infection of both A and B can be identified.

[2]Since updates are automatically displayed to all the friends by the OSN website, updates cannot appear in the previous scenarios

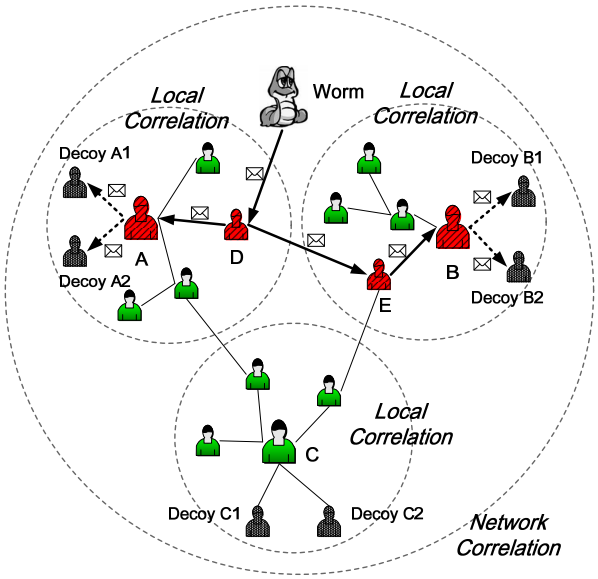


Figure 3: An example of two level correlation

3.5 Communication Module

The communication module acts as the interface of the detection system since it processes all necessary communications between the system and the OSN website. For example, it coordinates the communications between the configuration module with the administrator during system setting up. It receives propagation evidence from the OSN websites and passes them to the evidence collecting module. It also sends the worm infection alarm to the administrator on behalf of the worm detection module.

4. SYSTEM EVALUATION

In this section, we evaluate the OSN worm detection system on the real world social graph of Flickr [5]. There are 1,846,198 users and 22,613,981 friend links in the social graph, and the average friend number per user is 12.24. The data set is crawled from Flickr for a measurement study [25],

We evaluate the detection system with three objectives. The first objective is early warning. We define early warning in terms of the number of infected accounts by the time the worm is detected. This metric is borrowed from Internet worm detection. As suggested in [20], a worm detection system is deemed as an early warning system if the worm propagation is detected when less than 0.19% of all vulnerable hosts are infected. The second objective is to test the detection system under various worm propagation behaviors as well as under practical constraints such as user reluctance. The third objective is to examine the effectiveness of our system in worm containment with simple countermeasures provided.

To assess the practicability of the system design, we also discuss the implementation of the detection system in real world OSN websites.

4.1 Simulation Model

Our simulation model consists of four modules as shown in Figure 4. The initialization module takes the social graph

as input and output the set of selected users (the default size of the set is 500). For each selected user, this module adds two edges in the social graph from the vertex (representing the user) to the two new vertices (representing two decoy friends). It also creates and associates a SPEL with each selected user. Worm propagation starts from certain user(s). The propagation module models worm propagation by repeating two consecutive phases, namely sending worm messages (both messages and updates are referred as messages in evaluation) and infecting users. In the process of sending worm messages, our simulator chooses either to send worm messages to all the friends or to a fraction of friends randomly chosen from the friends list of an infected user. Each worm message is randomly selected from the a predefined worm messages set. We assume each recipient of worm messages gets infected with a probability of P_{user}. The probability is randomly assigned for each user and keeps constant in each worm propagation. The monitor module performs correlation tests on suspicious worm messages. Once a worm infection is identified, the post-processing module can output the infection statistic such as percentage of infected users.

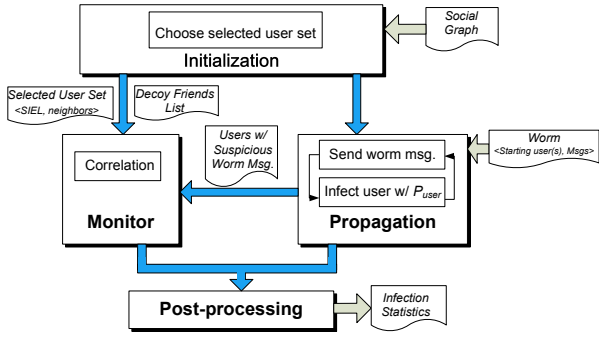


Figure 4: OSN Worm Simulation Model

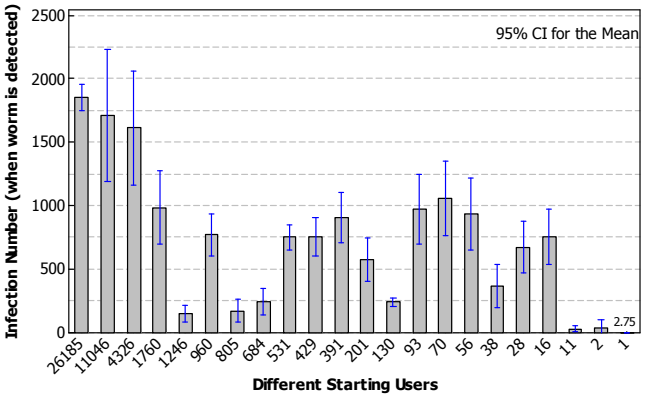
Since a few random variables are used in each propagation, we repeat each propagation for 100 times in our evaluation to reduce the impact of randomness. After that, we display the results as the mean values of these iterations as well as 95% confidence intervals of the means.

4.2 Early Warning Detection

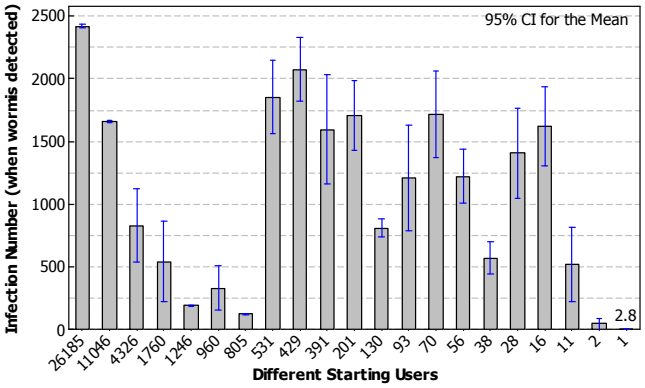
In this part of evaluation, we examine whether the detection system can achieve early warning. To this end, we test the detection system in the propagation of two representative OSN worms, namely Koobface worm and Mikeyy worm. A user account infected by Koobface worm sends different (customized) messages to the its two decoy friends. In Mikeyy worm propagation, an infected account delivers the same message to both decoys.

A crucial factor in worm propagation is the initially infected user account(s) (i.e., hitlist) because the total friend numbers of the initial account(s) can affect the worm propagation speed. To conduct a more comprehensive test, we choose 22 accounts from the Flickr dataset with different friend numbers (from 1 to 26,185, where 26,185 is the maximum number of friends) as initial infected user accounts and start worm propagation from these accounts in both worm cases. Then we measure the average infection numbers by the time these worms are detected and the results are showed in Table 1 and Figure 5.

Table 1 lists the average infection numbers on detection for both worms. The maximum infection number is 2420, which is only 0.13% of all the users. This indicates the detection system fulfills the early warning requirement. In addition, Table 1 shows the average infection number of Mikeyy worm is larger than that of Koobface worm. This is because the detection of Mikeyy worm infections requires both local correlation and network correlation. Next, we will use the Mikeyy worm propagation model for evaluation.



(a) Koobface



(b) Mikeyy

Figure 5: Infection Number versus Different Initial User Accounts

Figure 5 confirms that in general propagations starting from "popular" users can infect more user accounts. However, we also notice that in some cases worm propagation starting from less popular initial users can infect more user accounts than the propagations starting from more popular initial users. For example, in both worms, propagations starting from the user with 531 friends infects more accounts than propagations from the user with 1246 friends. The reason is that popular users are more likely to be included in the selected users set. If a user A, who has more friends than another user B, has decoy friends, the infection of A will be detected faster than infection of B. Therefore, a worm starting from A may infect less accounts than the same worm starting from B. Another reason, as demonstrated in [25], is that social graph tends to exhibit a tightly-connected "core"

Table 1: Infection Number on Detection for Koobface and Mikeyy Worms

Worm Type	Avg. Infection #	Max Infection #	Min Infection #
Koobface	700	1851	2.75
Mikeyy	1023	2420	2.8

of high-degree vertices connected with each other because of its scale-free metric and the positive assortatively coefficient. This property implies that when a "popular" user is infected, even if it does not belong to the selected user set, it is very likely to infect another "popular" user in a short time and this infection can be detected because the other "popular" user has decoy friends. This observation also justifies the heuristic we adopt in Algorithm 1, where we start with high-degree vertices.

4.3 Impact of Worm Behavior

In this part, we evaluate the detection system under behavior discrepancy of OSN worms. More specifically, we consider that an OSN worm randomly chooses a fraction of friends of an infected user as its propagation targets. We note current OSN worms infect all friends, making them easier to detect. Our evaluation is to show the effectiveness of our scheme against more intelligent worms. The impact of this behavior discrepancy is that some decoy friends may not receive propagation evidence even if the user accounts to which they are attached have been infected. To test our detection system under this assumption, we simulate worm propagations with usage of friend lists from 10% to 100%. For each percentage, worm propagation starts from the above 22 different accounts and the average results are showed in Figure 6.

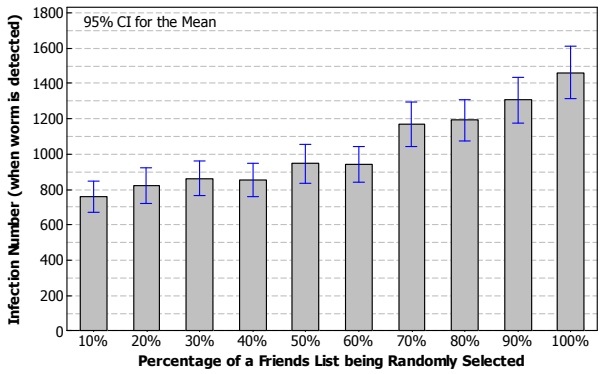


Figure 6: Infection Number versus Different Percentages of Friends lists

In Figure 6, all the worm propagations are detected when less than 1600 user accounts are infected. This indicates that the detection system can still achieve early warning when worms randomly choose targets from friend lists. We notice that even the worms using only 10% of the friend lists can still be detected. Hence, as long as worms have no knowledge about the identities of decoys, shrinking the size of target lists to reduce the probability of hitting decoys is ineffective to the attacker. On the other hand, if somehow worms spot some decoys, they can evade these known decoys (we will discuss the impact of this scenario in the next section). Another trend illustrated by Figure 6 is that when more friends are targeted, more accounts can be infected by worms. Therefore, OSN worms will tend to use as many contacts in the friends lists as possible if their goal is to enlarge infection.

4.4 Impact of Selected Users Set

As mentioned previously, not all the users in the selected users set are willing to accept decoy friends. Besides, we also consider the scenario where some of the decoys are spotted so that worms can avoid these decoys in propagation. To evaluate the detection system in these scenarios, we study worm propagations where only a part (randomly chosen) of the selected users set has decoy friends. We first choose a selected users set of 2000 users by Algorithm 1. After that, we randomly choose 100 to 2000 users from this set to assign decoy friends and then run worm propagations for each case. Again, for each set, worm propagation starts from the previously mentioned 22 different user accounts and the average infection numbers by the time of detection are illustrated in Figure 7.

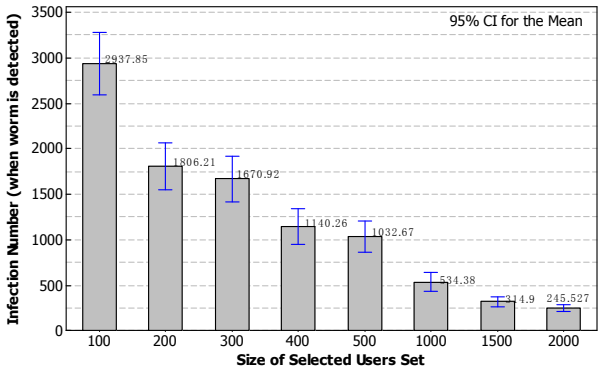


Figure 7: Infection Number versus the Size of Selected Users Set

Figure 7 clearly shows that a larger set of selected users with decoy friends can detect worms with fewer infected user accounts. For example, 1500 selected users set can detect worm propagations when only 314 user accounts are infected. However. when the size of the set is smaller or equal to 100, the infection number is larger than 3000, which implies that early warning may not be fulfilled with a selected users set at this size. On the other hand, the infection numbers are restrained less than 1000 when the set size is larger or equal to 500. This result shows the effectiveness of our detection system under the impact of a partial working surveillance network. It also indicates that the administrator of an OSN should encourage more users (if at all possible) to accept decoy friends, e.g., with incentives.

4.5 Containment Measures

Upon detecting a worm propagation, the system will notify the administrators of OSN websites. In addition, the

detection system can assist in suppressing worm propagations by adopting some simple countermeasures, such as warning the friends of infected users [11] (1-hop warning) by decoy friends or also warning the friends of friends (2-hops warning) if the privacy setting of a selected user allows the decoy friends to retrieve the friends information. In this study, we assume users will raise their vigilance after receiving the warning messages and for simplicity here we assume they will not be affected by the worm. The following results demonstrate the effectiveness of such warning mechanisms in worm containment.

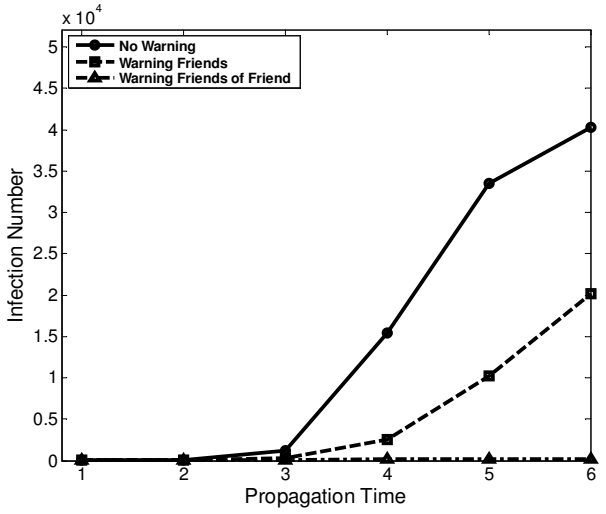


Figure 8: Worm Propagation versus Different Containment Measures

Figure 8 compares the infection numbers with and without warnings (both 1-hop and 2-hops warnings). The results are based on a setting where a set of 500 selected users are used to detect and warn other users. Figure 8 illustrates that warnings can effectively suppress the worm propagation. A 2-hop warning approach can limit the infected user number to a small value compared with the case in which no warning is issued. A 1-hop warning prevents nearly half of the users from being infected compared with the no warning case. These results with the simple countermeasures (e.g., 1-hop and 2-hops warnings) indicate the detection system is helpful in worm containment.

4.6 Example Implementation

We discuss a practical implementation of the worm detection system in detail based on a realworld OSN website, Facebook.The system can reside in a dedicated server. It has access to the Internet so that the server can visit Facebook and communicate with the administrator of Facebook through a secured channel (e.g., HTTPS or SSH). The modules run as programs on the server. All the modules have access to a database, which stores datasets such as the social graph of Facebook, selected users set, accounts of generated decoys and their login credentials, SPELs set, and identified worm evidence. The configuration module retrieves the social graph from the database and outputs selected users set to the database. It can also invoke an instance of a Web browser to register the decoy accounts on Facebook. The evidence collecting module is composed of an email client and a Web browser. By configuring the preference of each decoy account in Facebook, this program will be notified via emails about any suspicious evidence received by decoys. Upon the arrival of a notification email, the program of collecting module logs into the decoy account through the Web browser, retrieves the suspicious evidence and writes the evidence into the corresponding user's SPEL in the database. After that, the program invokes the worm detection program to process the evidence. The worm detection program can send notification to the administrator of Facebook via the secure channel if it identifies any infection based on the existing evidence in the database.

Although this implementation is based on Facebook, the features of Facebook used by the system (e.g., email notification) are widely supported by other OSN websites. Therefore, we believe this implementation can be adapted for other OSN websites with a slight modification. Moreover, our system can also be implemented in a distributed way. For example, the detection functionalities of the system may be distributed among decoys such that each decoy can perform its own worm detection by sharing suspicious evidence with other decoys.

5. LIMITATIONS

In this section, we discuss some limitations of our detection system.

One practical concern is raised from the scenario where normal messages spread in a worm-like pattern within an OSN website due to the outbreak of a large-scale event. For example, a breaking news can be broadcasted from one user to all his friends and keep multiplying in this way. There could be little difference between the propagation pattern of this news and an OSN worm. Unless there exists an approach to automatically distinguish the normal news from a worm message based on the content, little can we do to avoid the false alarms caused by this scenario. For example, if a posted link in a collected suspicious message is pointed to some well-known news websites, we may ignore it. Otherwise, which we believe is a very rare case, simple manual checking of the message content by an administrator will address the false alarm problem.

Another practical concern is in regards to the infection speed of OSN worms. Although one characteristic of OSN worms is fast infection, some of the worm infection cycles may be longer than the detection time window because users may respond to worm messages with different latencies. In this case, the only solution is to expand the time span of correlation. However, this could also degrade the performance of the detection system. As such, there is a trade-off between the computation resource and the demand of OSN websites.

6. RELATED WORK

In the area of Internet worm early detection, various detection strategies have been proposed. Gu *et al.* suggested an algorithm based on local victim information [20], in which they used destination-source correlation to capture the pattern in incoming and outgoing scanning traffics of a host before and after it is infected by a scan-based worm. They also looked for worm's anomalous scanning patterns, such as high scan rate to identify the outbreak of a worm. However,

their approach does not apply to OSN worm detection because no such scan traffic are present. Dagon *et al.* proposed a detection technique [18] using honeypot to monitor the entire infection process (infection cycle) rather than just the beginning and the end. They recorded memory event, network event and disk event to perform logistic analysis looking for correlation. Their approach requires no signature in advance and has the advantage of coping with polymorphic worms. However, lack of infection processes in OSN worms prevent applying their approach here. In fact, due to very limited worm activities, any approaches relying on detailed infection procedure is not suitable here.

Bu *et al.* suggested a worm detection scheme [17] based on the extraction of the alteration of arrival unsolicited scan rates in the early stage of worm propagation. Their work suggested a novel signal indicating the outbreak of an Internet worm, but this approach suffers from the problem of too many potential sources for false positive rate. Wagner *et al.* provided an entropy based worm detection algorithm [27]. They utilized entropy to quantify the difference of randomness observed in worm traffic and in normal traffic. The source IP address fields will be less random in worm traffic than in normal traffic since the scanning hosts' IP address were seen more than other hosts. Their strategy offered an alternative way to detect the propagation activities of an Internet worm. However, both of these approaches rely on the characteristics exhibited in worms' scanning traffic. For an OSN worm, no scan is necessary and the infection traffics are relatively simple compared to that of internet worm. Unlike packets with various attributes transmitted during the propagation of Internet worms, OSN worms merely generate messages. Moreover, there is no hierarchical structure in the organization of a social network. All peers are equal in the social graph, which means no auxiliary information is available for decisions of the location of a worm detector.

There are some other worm detection algorithms that are not based on scanning traffic. Wang *et al.* proposed an anomalous payload-based worm detection algorithm [28], a worm propagation can be identified if correlation of ingress and egress payload alerts is observed. In an OSN worm, the actual payload is downloaded in the browser, which cannot be observed by OSN websites. This is actually exploited by OSN worms to bypass any filtering based detection scheme deployed in OSN websites. However, one thing we can borrow from this work is the idea of correlation. As showed in this paper, we adopt correlation in worm activities to improve the detection accuracy.

7. CONCLUSIONS

In this paper, we design a system that can effectively detect the propagation of OSN worms. By exploiting the properties of OSNs, we construct a surveillance network embedded in the OSN websites using decoy friends. We also proposal an algorithm based on the heuristic derived from the topological properties of social graphs to keep the OSN websites under surveillance by monitoring only a few hundreds of users. We leverage both local and network correlations of worm propagation evidence in our detection system to achieve early warning detection.

Based on the real-world social graph of Flickr, our evaluation with two known worms, Koobface and Mikeyy, shows that the detection system can effectively detect OSN worm propagations when less than 0.13% of total user accounts are infected. Even taking user reluctance into consideration, our system can still achieve early warning detection. Moreover, the detection system is also demonstrated to applicable to worm containment by adopting some simple countermeasures. This can provide valuable assistance to OSN websites in fighting against worm propagations in future.

8. ACKNOWLEDGMENTS

We thank Alan Mislove for providing us with the social graph data set of Flickr. We also thank the reviewers for their valuable comments and suggestions. This work was supported by NSF CAREER 0643906.

9. REFERENCES

[1] 56th variant of the koobface worm detected. `http://blogs.zdnet.com/security/?p=3414`.

[2] Cross-site scripting worm floods myspace.

[3] Facebook security threats. `http://www.facebook.com/security?v=app\_4949752878\&viewas=1661798617\&ref=search`.

[4] Facebook statistics. `http://www.facebook.com/press/info.php?statistics`.

[5] Flickr. `http://www.flickr.com/`.

[6] Koobface virus hits facebook. `http://news.cnet.com/koobface-virus-hits-facebook/`.

[7] Linkedin: About us. `http://press.linkedin.com/about`.

[8] Msrt august top detection reports. `http://blogs.technet.com/mmpc/archive/2009/08/27/msrt-august-top-detection-reports.aspx`.

[9] Myspace fact sheet. `http://www.myspace.com/pressroom?url=/fact+sheet/`.

[10] New worms target both myspace and facebook users. `http://www.kaspersky.com/news?id=207575670`.

[11] Steps you should take. `http://www.facebook.com/security`.

[12] Teen claims responsibility for disrupting twitter.

[13] Twitter users number.

[14] W32.koobface.a. `http://www.symantec.com/security\_response/writeup.jsp?docid=2008-080315-0217-99\&tabid=2`.

[15] M. Boguna, R. Pastor-Satorras, and A. Vespignani. Epidemic spreading in complex networks with degree correlations. *Lecture Notes in Pgysics: Statistical Mechanics of Complex Networks*, 625:127–147, 2003.

[16] G. Brown, T. Howe, M. Ihbe, A. Prakash, and K. Borders. Social networks and context-aware spam. In *Proceedings of the 2008 ACM Conference on Computer Supported Cooperative Work*, San Diego, CA, 2008.

[17] T. Bu, A. Chen, S. Vander Wiel, and T. Woo. Design and evaluation of a fast and robust worm detection algorithm. In *Proceedings of the 25th IEEE International Conference on Computer Communications*, Barcelona, Catalunya, Spain, 2006.

[18] D. Dagon, X. Qin, G. Gu, W. Lee, J. Grizzard, J. Levine, and H. Owen. Honeystat: Local worm detection using honeypots. In *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection*, Sophia Antipolis, French Riviera, France, 2004.

[19] Y. Fu, X. Wang, and S. Li. Construction k-dominating set with multiple relaying technique in wireless mobile ad hoc networks. In *Proceedings of the 2009 WRI International Conference on Communications and Mobile Computing*, Kunming, Yunnan, China, 2009.

[20] G. Gu, M. Sharif, X. Qin, D. Dagon, W. Lee, and G. Riley. Worm detection, early warning and response based on local victim information. In *Proceedings of the 20th Annual Computer Security Applications Conference*, Tucson, AZ, 2004.

[21] R. Kumar, J. Novak, and A. Tomkins. Structure and evolution of online social networks. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Philadelphia, PA, 2006.

[22] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics-Doklady*, 10:707–709, 1966.

[23] D. Lin. An information-theoretic definition of similarity. In *Proceedings of the 15th International Conference on Machine Learning*, Madison, WI, 1998.

[24] B. Livshits and W. Cui. Spectator: Detection and containment of javascript worms. In *Proceedings of the 2008 USENIX Annual Technical Conference*, Boston, MA, 2008.

[25] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, San Diego, CA, 2007.

[26] Y. Moreno, R. Pastor-Satorras, and A. Vespignani. Epidemic outbreaks in complex heterogeneous networks. *The European Physical Journal B - Condensed Matter and Complex Systems*, 26:521–529, 2002.

[27] A. Wagner and B. Plattner. Entropy based worm and anomaly detection in fast ip networks. In *Proceedings of the 14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise*, Linkoping, Sweden, 2005.

[28] K. Wang, G. Cretu, and S. J. Stolfo. Anomalous payload-based worm detection and signature generation. In *Proceedings of the 8th International Symposium of Recent Advances in Intrusion Detection*, Seattle, WA, 2005.

[29] J. Wu, M. Cardei, F. Dai, and S. Yang. Extended dominating set and its applications in ad hoc networks using cooperative communication. *IEEE Trans. Parallel Distrib. Syst.*, 17(8):851–864, 2006.

[30] M. Xie, Z. Wu, and H. Wang. Honeyim: Fast detection and suppression of instant messaging malware in enterprise-like networks. In *Proceedings of the 23th Annual Computer Security Applications Conference*, Miami Beach, FL, 2007.

[31] C. C. Zou, D. Towsley, and W. Gong. Modeling and simulation study of the propagation and defense of internet e-mail worms. *IEEE Trans. Dependable Secur. Comput.*, 4(2):105–118, 2007.

Who is Tweeting on Twitter: Human, Bot, or Cyborg?

Zi Chu, Steven Gianvecchio and Haining Wang
Department of Computer Science
The College of William and Mary
Williamsburg, VA 23187, USA
{zichu, srgian, hnw}@cs.wm.edu

Sushil Jajodia
Center for Secure Information Systems
George Mason University
Fairfax, VA 22030, USA
jajodia@gmu.edu

ABSTRACT

Twitter is a new web application playing dual roles of online social networking and micro-blogging. Users communicate with each other by publishing text-based posts. The popularity and open structure of Twitter have attracted a large number of automated programs, known as bots, which appear to be a double-edged sword to Twitter. Legitimate bots generate a large amount of benign tweets delivering news and updating feeds, while malicious bots spread spam or malicious contents. More interestingly, in the middle between human and bot, there has emerged cyborg referred to either bot-assisted human or human-assisted bot. To assist human users in identifying who they are interacting with, this paper focuses on the classification of human, bot and cyborg accounts on Twitter. We first conduct a set of large-scale measurements with a collection of over 500,000 accounts. We observe the difference among human, bot and cyborg in terms of tweeting behavior, tweet content, and account properties. Based on the measurement results, we propose a classification system that includes the following four parts: (1) an entropy-based component, (2) a machine-learning-based component, (3) an account properties component, and (4) a decision maker. It uses the combination of features extracted from an unknown user to determine the likelihood of being a human, bot or cyborg. Our experimental evaluation demonstrates the efficacy of the proposed classification system.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and Protection*

General Terms

Security

Keywords

Automatic Identification, Bot, Cyborg, Twitter

1. INTRODUCTION

Twitter is a popular online social networking and micro-blogging tool, which was released in 2006. Remarkable simplicity is its distinctive feature. Its community interacts via publishing text-based posts, known as *tweets*. The tweet size is limited to 140 characters. Hashtag, namely words or phrases prefixed with a # symbol, can group tweets by topic. For example, #Haiti and #Super Bowl are the two trending hashtags on Twitter in January 2010. Symbol @ followed by a username in a tweet enables the direct delivery of the tweet to that user. Unlike most online social networking sites (i.e., Facebook and MySpace), Twitter's user relationship is directed and consists of two ends, friend and follower. In the case where the user A adds B as a friend, A is a *follower* of B while B is a *friend* of A. In Twitter terms, A follows B. B can also add A as his friend (namely, following back or returning the follow), but is not required. From the standpoint of information flow, tweets flow from the source (author) to subscribers (followers). More specifically, when a user posts tweets, these tweets are displayed on both the author's homepage and those of his followers.

Since 2009, Twitter has gained increasing popularity. As reported in June 2010, Twitter is attracting 190 million visitors per month and generating 65 million Tweets per day [30]. It ranks the 12th on the top 500 site list according to Alexa [5]. In November 2009, Twitter emphasized its value as a news and information network by changing the question above the tweet input dialog box from "What are you doing" to "What's happening". To some extent, Twitter is in the transition from a personal micro-blogging site to an information publish venue. Many traditional industries have used Twitter as a new media channel. We have witnessed successful Twitter applications in business promotion [1], customer service [3], political campaigning [2], and emergency communication [21, 35].

The growing user population and open nature of Twitter have made itself an ideal target of exploitation from automated programs, known as bots. Like existing bots in other web applications (i.e., Internet chat [14], blogs [34] and online games [13]), bots have been common on Twitter. Twitter does not inspect strictly on automation. It only requires the recognition of a CAPTCHA image during registration. After gaining the login information, a bot can perform most human tasks by calling Twitter APIs. More interestingly, in the middle between humans and bots have emerged cyborgs, which refer to either bot-assisted humans or human-assisted bots. Cyborgs have become common on Twitter. After a human registers an account, he may set automated programs (i.e., RSS feed/blog widgets) to post tweets during his absence. From time to time, he participates to tweet and interact with friends. Cyborgs interweave characteristics of both humans and bots.

Automation is a double-edged sword to Twitter. On one hand, legitimate bots generate a large volume of benign tweets, like news and blog updates. This complies with the Twitter's goal of becoming a news and information network. On the other hand, malicious bots have been greatly exploited by spammers to spread spam or malicious contents. These bots randomly add users as their friends, expecting a few users to follow back[1]. In this way, spam tweets posted by bots display on users' homepages. Enticed by the appealing text content, some users may click on links and get redirected to spam or malicious sites[2]. If human users are surrounded by ma-

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

[1]Some advanced bots target potential users by keyword search.

[2]Due to the tweet size limit, it is very common to use link shortening service on Twitter, which converts an original link to a short one (i.e., http://bit.ly/dtUm5Q). The link illegibility favors bots to

licious bots and spam tweets, their twittering experience deteriorates, and eventually the whole Twitter community will be hurt. The objective of this paper is to characterize the automation feature of Twitter accounts, and to classify them into three categories, human, bot, and cyborg, accordingly. This will help Twitter manage the community better and help human users recognize who they are tweeting with.

In the paper, we first conduct a series of measurements to characterize the differences among human, bot, and cyborg in terms of tweeting behavior, tweet content, and account properties. By crawling Twitter, we collect over 500,000 users and more than 40 million tweets posted by them. Then we perform a detailed data analysis, and find a set of useful features to classify users into the three classes. Based on the measurement results, we propose an automated classification system that consists of four major components: (1) the entropy component uses tweeting interval as a measure of behavior complexity, and detects the periodic and regular timing that is an indicator of automation; (2) the machine-learning component uses tweet content to check whether text patterns contain spam or not[3]; (3) the account properties component employs useful account properties, such as tweeting device makeup, URL ration, to detect deviations from normal; (4) the decision maker is based on Linear Discriminant Analysis (LDA), and it uses the linear combination of the features generated by the above three components to categorize an unknown user as human, bot or cyborg. We validate the efficacy of the classification system through our test dataset. We further apply the system to classify the entire dataset of over 500,000 users collected, and speculate the current composition of Twitter user population based on our classification results.

The remainder of this paper is organized as follows. Section 2 covers related work on Twitter and online social networks. Section 3 details our measurements on Twitter. Section 4 describes our automatic classification system on Twitter. Section 5 presents our experimental results on classification of humans, bots, and cyborgs on Twitter. Finally, Section 6 concludes the paper.

2. RELATED WORK

Twitter has been widely used since 2006, and there are some related literature in twittering [24, 25, 43]. To better understand micro-blogging usage and communities, Java et al. [24] studied over 70,000 Twitter users and categorized their posts into four main groups—daily chatter (e.g., "going out for dinner"), conversations, sharing information or URLs, and reporting news—and further classified their roles by link structure into three main groups—information source, friends, and information seeker. Their work also studied (1) the growth of Twitter, showing a linear growth rate; (2) its network properties, showing the evidence that the network is scale-free like other social networks [27]; and (3) the geographical distribution of its users, showing that most Twitter users are from the US, Europe, and Japan. Krishnamurthy et al. [25] studied a group of over 100,000 Twitter users and classified their roles by follower-to-following ratios into three groups: (1) broadcasters, which have a large number of followers; (2) acquaintances, which have about the same number on either followers or following; and (3) miscreants and evangelists (e.g., spammers), which follow a large number of other users but have few followers. Their work also examined the growth of Twitter, revealing a greater than linear growth rate. In a more recent work, Yardi et al. [43] investigated spam on Twitter. According to their observations, spammers send more messages than legitimate users, and are more likely to follow other spammers than legitimate users. Thus, a high follower-to-following ratio is a sign of spamming behavior. Kim et al. [10] analyzed Twitter lists as a potential source for discovering latent characters and interests of users. A Twitter list consists of multiple users and their tweets. Their research indicated that words extracted from each list are representative of all the members in the list even if the words are not used by the members. It is useful for targeting users with specific interests.

Compared to previous measurement studies on Twitter, our work covers a much larger group of Twitter users (more than 500,000) and differs in how we link the measurements to automation, i.e., whether posts are from humans, bots, or cyborgs. While some similar metrics are used in our work, such as follower-to-following ratio, we also introduce some metrics, including entropy of tweet intervals, which are not employed in previous research. In addition to network-related studies, several previous works focus on socio-technological aspects of Twitter [21, 23, 32, 35, 45], such as its use in the workplace or during major disaster events.

Twitter is a social networking service, so our work is also related to recent studies on social networks, such as Flickr, LiveJournal, Facebook, MySpace, and YouTube [6, 7, 27]. In [27], with over 11 million users of Flickr, YouTube, LiveJournal, and Orkut, Mislove et al. analyzed link structure and uncovered the evidence of power-law, small-world, and scale-free properties. In [7], Cha et al. examined the propagation of information through the social network of Flickr. Their work shows that most pictures are propagated through the social links (i.e., links received from friends rather than through searches or external links to Flickr content) and the propagation is very slow at each hop. As a result of this slow propagation, a picture's popularity is often localized in one network and grows slowly over a period of months or even years. In [6], Cha et al. analyzed video popularity life-cycles, content aliasing, and the amount of illegal content on YouTube, a popular video sharing service. While YouTube is designed to share large content, i.e., videos, Twitter is designed to share small content, i.e., text messages. Unlike other social networking services, like Facebook or YouTube, Twitter is a micro-content social network, with messages being limited to 140 characters.

As Twitter is a text-based message system, it is natural to compare it with other text-based message systems, such as instant messaging or chat services. Twitter has similar message length (140 characters) to instant messaging and chat services. However, Twitter lacks "presence" (users show up as online/offline for instant messaging services or in specific rooms for chat) but offers (1) more access methods (web, SMS, and various APIs) for reading or posting and (2) more persistent content. Similar to Twitter, instant messaging and chat services also have problems with bots and spam [14, 40]. To detect bots in online chat, Gianvecchio et al. [14] analyzed humans and bots in Yahoo! chat and developed a classification system to detect bots using entropy-based and machine-learning-based classifiers, both of which are used in our classification system as well. In addition, as Twitter is text-based, email spam filtering techniques are also relevant [17, 41, 44]. However, Twitter posts are much shorter than emails and spaced out over longer periods of time than for instant messages, e.g., hours rather than minutes or seconds.

Twitter also differs from most other network services in that automation, e.g., message feeds, is a major feature of legitimate Twitter usage, blurring the lines between bot and human. Twitter users can be grouped into four categories: humans, bots, bot-assisted humans, and human-assisted bots. The latter two, bot-assisted humans and human-assisted bots, can be described as cyborgs, a mix between bots and humans [42].

3. MEASUREMENT

In this section, we first describe the data collection of over 500,000 Twitter users. Then, we detail our observation of user behaviors and account properties, which are pivotal to automatic classification.

3.1 Data Collection

Here we present the methodology used to crawl the Twitter network and collect detailed user information. Twitter has released a set of API functions [39] that support user information collection. Thanks to Twitter's courtesy of including our test account to its white list, we can make API calls up to 20,000 per hour. This eases our data collection. To diversify our data sampling, we employ two methods to collect the dataset covering more than 500,000 users. The first method is Depth-First Search (DFS) based crawling. The reason we choose DFS is that it is a fast and uniformed algorithm for traversing a network. Besides, DFS traversal implicitly includes the information about network locality and clustering. Inspired by [15, 18], we randomly select five users as seeds. For

allure users.

[3]Spam is a good indicator of automation. Most spam messages are generated by bots, and very few are manually posted by humans.

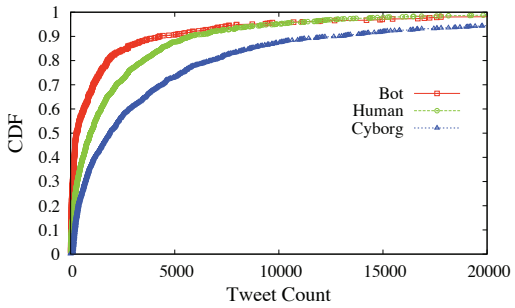


Figure 1: CDF of Tweet Count

each reached user, we record its follower list. Taking the following direction, the crawler continues with the depth constraint set as three. We customize our crawler with a core module of PHP cURL. Ten crawler processes work simultaneously for each seed. After a seed is finished, they move to the next. The crawl duration lasts four weeks from October 20th to November 21st, 2009, and 429,423 users are logged.

Similar to the work in [25] and [43], we also use the public timeline API to collect the information of active users, increasing the diversity of the user pool. Twitter constantly posts the twenty most recent tweets in the global scope. The crawler calls the timeline API to collect the authors of the tweets included in the timeline. Since the Twitter timeline frequently updates, the crawler can repeatedly call the timeline API. During the same time window of the DFS crawl, this method contributes 82,984 users to the dataset. We totally collect 512,407 users on Twitter combining both methods.

3.2 Ground Truth Creation

To develop an automatic classification system, we need a training data set that contains known samples of human, bot, and cyborg. Among collected data, we randomly choose different samples and classify them by manually checking their user logs and homepages. The training set includes one thousand users per class of human, bot and cyborg, and thus in total there are three thousand classified samples. A test set of three thousand samples is created in a similar way. Both sets serve as the ground truth dataset, containing 8,350,095 tweets posted by the sampled users in their account lifetime[4], from which we can extract useful features for classification, such as tweeting behaviors and text patterns.

Our log-based classification follows the principle of the Turing test [36]. The standard Turing tester communicates with an unknown subject for five minutes, and decides whether it is a human or machine. Classifying Twitter users is actually more challenging than it appears to be. For many users, their tweets are less likely to form a relatively consistent context. For example, a series of successive tweets may be hardly relevant. The first tweet is the user status, like "watching a football game with my buds." The second tweet is an automatic update from his blog. The third tweet is a news report RSS feed in the format of article title followed by a shortened URL.

For every account, the following classification procedure is executed. We thoroughly observe the log, and visit the user's homepage (http://twitter.com/username) if necessary. We carefully check tweet contents, visit URLs included in tweets (if any), and decide if redirected web pages are related with their original tweets and if they contain spam or malicious contents. We also check other properties, like tweeting devices, user profile, and the numbers of followers and friends. Given a long sequence of tweets (usually we check 60 or more if needed), the user is labeled as a human if we can obtain some evidence of original, intelligent, specific and human-like contents. In particular, a human user usually records what he is doing or how he feels about something on Twitter, as he uses Twitter as a micro-blogging tool to display himself and interact with friends. For example, he may write a post like "I just saw Yankees lost again today. I think they have to replace the starting pitcher for tomorrow's game." The content carries intelligence and originality. Specificity means that the tweet content is expressed in relatively unambiguous words with the presence of consciousness [36]. For instance, in reply to a tweet like "How you like iPad?", a specific response made by human may be "I like its large touch screen and embedded 3G network". On the other hand, a generic reply could be "I like it".

The criteria for identifying a bot are listed as follows. The first is the lack of intelligent or original content. For example, completely retweeting tweets of others or posting adages indicates a lack of originality. The second is the excessive automation of tweeting, like automatic updates of blog entries or RSS feeds. The third is the abundant presence of spam or malicious URLs (i.e., phishing or malware) in tweets or the user profile. The fourth is repeatedly posting duplicate tweets. The fifth is posting links with unrelated tweets. For example, the topic of the redirected web page does not match the tweet description. The last is the aggressive following behavior. In order to gain attention from human users, bots do mass following and un-following within a short period of time. Cyborgs are either human-assisted bots or bot-assisted humans. The criterion for classifying a cyborg is the evidence of both human and bot participation. For example, a typical cyborg account may contain very different types of tweets. A large proportion of tweets carry contents of human-like intelligence and originality, while the rest are automatic updates of RSS feeds. It represents a usage model, in which the human uses his account from time to time while the Twitter widget constantly runs on his desktop and posts RSS feeds of his favorite news channel. Lastly, the uncertain category is for non-English users and those without enough tweets to classify. The samples that are difficult and uncertain to classify fall into this category, and are discarded. Some Twitter accounts are set as "private" for privacy protection, and their web pages are only visible to their friends. We do not include such type of users in the classification either, because of their inaccessibility.

3.3 Data Analysis

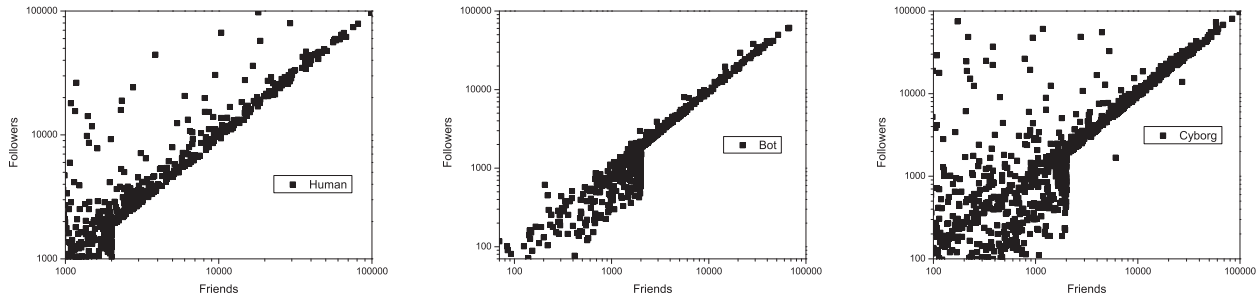
As mentioned before, Twitter API functions support detailed user information query, ranging from profile, follower and friend lists to posted tweets. In the above crawl, for each user visited, we call API functions to collect abundant information related with user classification. Most information is returned in the format of XML or JSON. We develop some toolkits to extract useful information from the above well-organized data structures. Our measurement results are presented in the question-answer format.

Q1. Does automation generate more tweets? To answer Question 1, we measure the number of tweets posted in a user's lifetime [5]. Figure 1 shows the cumulative distribution function (CDF) of the tweet counts, corresponding to the human, bot and cyborg category. It is clear that cyborg posts more tweets than human and bot. A large proportion of cyborg accounts are registered by commercial companies and websites as a new type of media channel and customer service. Most tweets are posted by automated tools (i.e., RSS feed widgets, Web 2.0 integrators), and the volume of such tweets is considerable. Meanwhile, those accounts are usually maintained by some employees who communicate with customers from time to time. Thus, the high tweet count in the cyborg category is attributed to the combination of both automatic and human behaviors in a cyborg. It is surprising that bot generates fewer tweets than human. We check the bot accounts, and find out the following fact. In its active period, bot tweets more frequently than human. However, bots tend to take long-term hibernation. Some are either suspended by Twitter due to extreme or aggressive activities, while the others are in incubation and can be activated to form bot legions.

Q2. Do bots have more friends than followers? A user's tweets can only be delivered to those who follow him. A common strategy shared by bots is following a large number of users (either targeted with purpose or randomly chosen), and expecting some of them will follow back. Figure 2 shows the scatter plots of the numbers

[4]4,431,923 tweets in the training set, and 3,918,172 tweets in the test set.

[5]It is the duration from the time when his account was created to the time when our crawler visited it.



(a) Human (b) Bot (c) Cyborg

Figure 2: Numbers of Followers and Friends

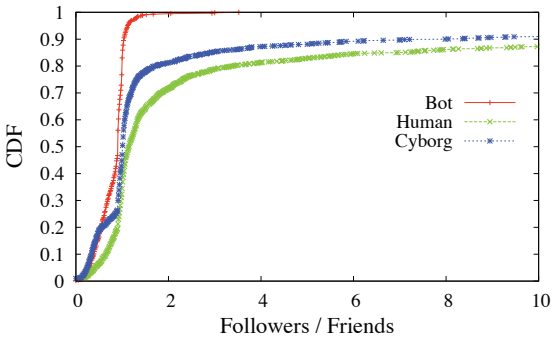


Figure 3: CDF of ratio of Followers over Friends

of followers and friends for the three categories. For better illustration, the scale is chopped and a small amount of extraordinary points are not included. In Figure 2, there are three different groups of users: group I where the number of one's followers is clearly greater than the number of its friends; group II where the situation is reverse; and group III where the nodes stick around the diagonal.

In the human category, as shown in Figure 2(a), the majority of the nodes belong to group III, implying that the number of their followers is close to that of their friends. This result complies with [27], revealing that human relationships are typically reciprocal in social networks. Meanwhile, there are quite a few nodes belonging to group I with far more followers than friends. They are usually accounts of celebrities and famous organizations. They generate interesting media contents and attract numerous subscribers. For example, the singer Justin Timberlake has 1,645,675 followers and 39 friends (the ratio is 42,197-to-1).

In the bot category, many nodes belong to group II, as shown in Figure 2(b). Bots add many users as friends, but few follow them back. Unsolicited tweets make bots unpopular among the human world. However, for some bots, the number of their followers is close to that of their friends. This is due to the following reason. Twitter imposes a limit on the ratio of followers over friends to suppress bots. Thus, some more advanced bots unfollow their friends if they do not follow back within a certain amount of time. Those bots cunningly keep the ratio close to 1. Figure 3 shows the ratio of followers over friends for the three categories. The human ratio is the highest, whereas the bot ratio is the lowest.

Q3. Are there any other temporal properties of Twitter users helpful for differentiation among human, bot, and cyborg? Many research works like [11] and [9] have shown the weekly and diurnal access patterns of humans in the Internet. Figures 4(a) and 4(b) present the tweeting percentages of the three different categories on daily and hourly bases, respectively. The weekly behavior of Twitter users shows clear differences among the three categories. While humans are more active during the regular workdays, from Monday to Friday, and less active during the weekend, Saturday and Sunday, bots have roughly the same activity level every day of the week. Interestingly, cyborgs are the most active ones on Monday and then slowly decrease their tweeting activities during the week; on Saturday cyborgs reach their lowest active point but somehow bounce back a bit on Sunday. Such a cyborg activity trend is mainly caused by their message feeds and the high level of news and blog activities at the start of a week. Similarly, the hourly behavior of human is more active during the daytime, which mostly overlaps with office hours. The bot activity is nearly even except a little drop in the deep of night. Some more advanced bots have the setting of "only tweet from a time point to another," which helps save API calls [37]. Thus, they can tweet more in the daytime to better draw the attention of humans.

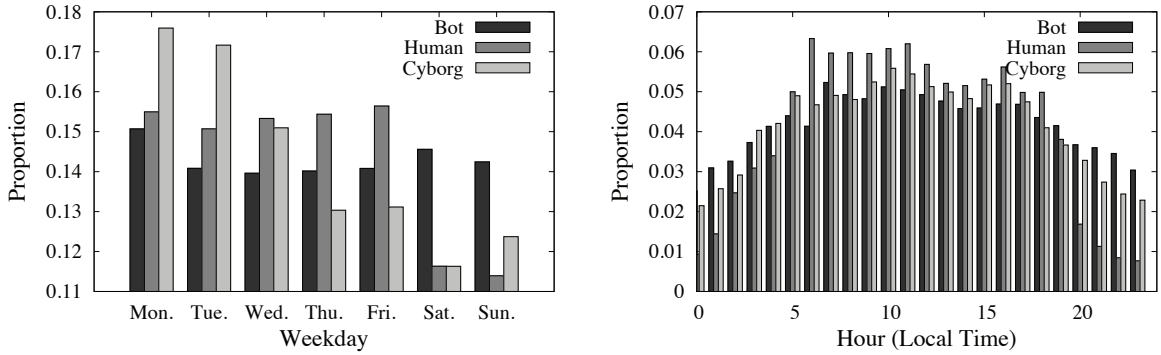
Figure 5 shows account registration dates grouped by quarter. We have two observations from the figure. First, the majority of accounts (80.0% of humans, 94.8% of bots, and 71.1% of cyborgs) were registered in 2009. It confirms the dramatic growth of Twitter in 2009. Second, we do not find any bot or cyborg in our ground truth dataset earlier than March, 2007. However, human registration has continued increasing since Twitter was founded in 2006. Thus, old accounts are less likely to be bots.

Q4. How do users post tweets? manually or via auto piloted tools? Twitter supports a variety of channels to post tweets. The device name appears below a tweet prefixed by "from." Our whole dataset includes 41,991,545 tweets posted by 3,648 distinct devices. The devices can be roughly divided into the following four categories. (1) Web, a user logs into Twitter and posts tweets via the website. (2) Mobile devices, there are some programs exclusively running on mobile devices to post tweets, like Txt for text messages, Mobile web for web browsers on handheld devices, TwitterBerry for BlackBerry, and twidroid for Android mobile OS. (3) Registered third-party applications, many third-parties have developed their own applications using Twitter APIs to tweet, and registered them with Twitter. From the application standpoint, we can further categorize this group into sub groups including website integrators (twitpic, bit.ly, Facebook), browser extensions (Tweetbar and Twitterfox for Firefox), desktop clients (TweetDeck and Seesmic Desktop), and RSS feeds/blog widgets (twitterfeed and Twitter for Wordpress). (4) APIs, for those third-party applications not registered or certificated by Twitter, they appear as "API" in Twitter.

Figure 6 shows the makeup of the above tweeting device categories. Among them, the website of Twitter is the most widely used and generates nearly half of the tweets (46.78%), followed by third-party devices (40.18%). Mobile devices and unregistered API tools contribute 6.81% and 6.23%, respectively. Table 1 lists the top ten devices used by the human, bot, and cyborg categories, and the whole dataset[6].

More than half of the human tweets are manually posted via

[6]The whole dataset contains around 500,000 users, and the human, bot and cyborg categories equally contain 1,000 users in the training dataset.



(a) Tweets by Day of Week (b) Hourly Tweets

Figure 4: Tweets Posted

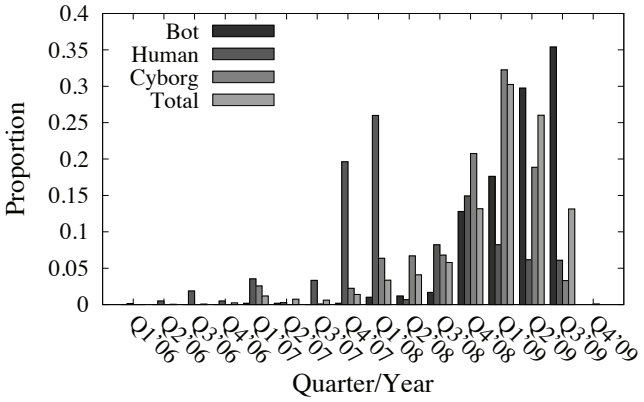


Figure 5: Account Registration Date (Grouped by Quarter)

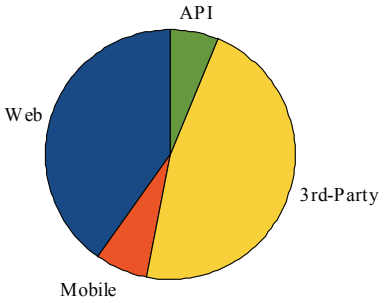


Figure 6: Tweeting Device Makeup

the Twitter website. The rest of top devices are mobile applications (Tweetie, UberTwitter, Mobile web, Txt, TwitterBerry) and desktop clients (TweetDeck, Echofon and Seesmic). In general, tweeting via such devices requires human participation. In contrast, the top tools used by bots are mainly auto piloted, and 42.39% of bot tweets are generated via unregistered API-based tools. Bots can abuse APIs to do almost everything they want on Twitter, like targeting users with keywords, following users, unfollowing those who do not follow back, or posting prepared tweets. Twitterfeed, RSS2Twitter, and Proxifeed are RSS feed widgets that automatically pipeline information (usually in the format of the page title followed by the URL) to Twitter via RSS feeds. Twitter Tools and Twitme for WordPress are popular WordPress plug-ins that integrate blog updates to Twitter. Assetize is an advertising syndicator mainly targeting at Twitter, and twitRobot is a bot tool that automatically follows other users and posts tweets. All these tools only require minimum human participation (like importing Twitter account information, or setting RSS feeds and update frequency), and thus indicate great automation.

Overall, humans tend to tweet manually and bots are more likely to use auto piloted tools. Cyborgs employ the typical human and bot tools. The cyborg group includes many human users who access their Twitter accounts from time to time. For most of the time when they are absent, they leave their accounts to auto piloted tools for management.

Q5. Do bots include more external URLs than humans? In our measurement, we find out that, most bots tend to include URLs in tweets to redirect visitors to external web pages. For example, spam bots are created to spread unsolicited commercial information. Their topics are similar to those in email spam, including online marketing and affiliate programs, working at home, selling fake luxury brands or pharmaceutical products[7]. However, the tweet size is up to 140 characters, which is rather limited for spammers to express enough text information to allure users. Basically, a spam tweet contains an appealing title followed by an external URL. Figure 7 shows the external URL ratios (namely, the number of external URLs included in tweets over the number of tweets posted by an account) for the three categories, among which the URL ratio of bot is highest. Some tweets by bots even have more than one URL[8]. The URL ratio of cyborg is very close to the bot's level. A large number of cyborgs integrate RSS feeds and blog updates, which take the style of webpage titles followed by page links. The URL ratio of human is much lower, on average it is only 29%. When a human tweets what is he doing or what is happening around him, he mainly uses text and does not often link to web pages.

Q6. Are users aware of privacy and identity protection on Twitter? Twitter provides a `protected` option to protect user privacy. If it is set as true, the user's homepage is only visible to his friends. However, the option is set as false by default. In our dataset of over 500,000 users, only 4.9% of them are protected users. Twitter also verifies some accounts to authenticate users' real identities. More and more celebrities and famous organizations have applied for verified accounts. For example, Bill Gates has his verified Twitter account at `http://twitter.com/billgates`. However, in our dataset, only 1.8% of users have verified accounts.

[7]A new topic is attracting more followers on Twitter. It follows the style of pyramid sales by asking newly joined users to follow existing users in the spam network.

[8]Many such accounts belong to a type of bot that always appends a spam link to tweets it re-tweets.

Table 1: Top 10 Tweeting Devices

Rank	Human	Bot	Cyborg	All
#1	Web (50.53%)	API (42.39%)	Twitterfeed (31.29%)	Web (46.78%)
#2	TweetDeck (9.19%)	Twitterfeed (26.11%)	Web (23.00%)	TweetDeck (9.26%)
#3	Tweetie (6.23%)	twitRobot (13.11%)	API (6.94%)	Twitterfeed (7.83%)
#4	UberTwitter (3.64%)	RSS2Twitter (2.66%)	Assetize (5.74%)	API (6.23%)
#5	Mobile web (3.02%)	Twitter Tools (1.24%)	HootSuite (5.22%)	Echofon (2.80%)
#6	Txt (2.56%)	Assetize (1.17%)	WP to Twitter (2.40%)	Tweetie (2.50%)
#7	Echofon (2.22%)	Proxifeed (1.08%)	TweetDeck (1.54%)	Txt (2.13%)
#8	TwitterBerry (2.10%)	TweetDeck (0.99%)	UberTwitter (1.19%)	HootSuite (2.10%)
#9	Twitterrific (1.93%)	bit.ly (0.91%)	RSS2Twitter (1.18%)	UberTwitter (1.71%)
#10	Seesmic(1.64%)	Twitme for WordPress (0.84%)	Twaitter (0.86%)	Mobile web (1.53%)

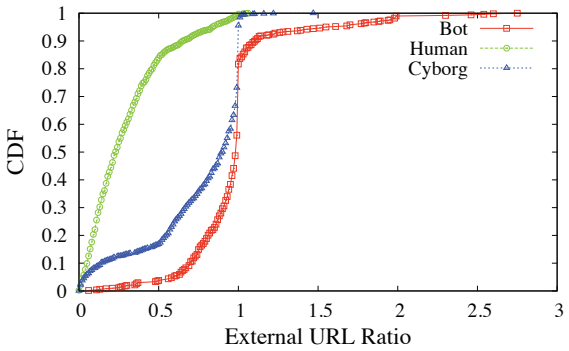


Figure 7: External URL ratio in tweets

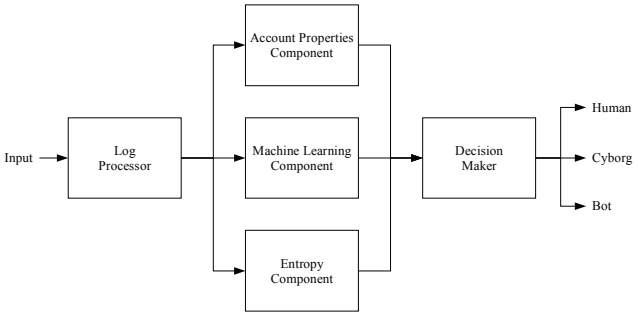


Figure 8: Classification System

4. CLASSIFICATION

This section describes our automated system for classification of Twitter users. The system classifies Twitter users into three categories: human, bot, and cyborg. The system consists of several components: the entropy component, the machine learning component, the account properties component, and the decision maker. The high-level design of our Twitter user classification system is shown in Figure 8. The entropy component uses corrected conditional entropy to detect periodic or regular timing, which is a sign of automation. The machine learning component uses a variant of Bayesian classification to detect text patterns of known spam on Twitter. The account properties component uses account-related properties to catch bot deviation from the normal human distribution. Lastly, the decision maker uses LDA to analyze the features identified by the other three components and makes a decision: human, cyborg, or bot.

4.1 Entropy Component

The entropy component detects periodic or regular timing of the messages posted by a Twitter user. On one hand, if the entropy or corrected conditional entropy is low for the inter-tweet delays, it indicates periodic or regular behavior, a sign of automation. More specifically, some of the messages are posted via automation, i.e., the user may be a potential bot or cyborg. On the other hand, a high entropy indicates irregularity, a sign of human participation.

4.1.1 Entropy Measures

The entropy rate is a measure of the complexity of a process [8]. The behavior of bots is often less complex than that of humans [12,22], which can be measured by entropy rate. A low entropy rate indicates a regular process, whereas a high entropy rate indicates a random process. A medium entropy rate indicates a complex process, i.e., a mix of order and disorder [20].

The entropy rate is defined as either the average entropy per random variable for an infinite sequence or as the conditional entropy of an infinite sequence. Thus, as real datasets are finite, the conditional entropy of finite sequences is often used to estimate the entropy rate. To estimate the entropy rate, we use the corrected conditional entropy [28]. The corrected conditional entropy is defined as follows.

A random process $X = \{X_i\}$ is defined as a sequence of random variables. The entropy of such a sequence of random variables is defined as:

$$H(X_1, ..., X_m) = -\sum_{X_1,...,X_m} P(x_1, ..., x_m) \log P(x_1, ..., x_m), \tag{1}$$

where $P(x_1, ..., x_m)$ is the joint probability $P(X_1 = x_1, ..., X_m = x_m)$.

The conditional entropy of a random variable given a previous sequence of random variables is:

$$H(X_m \mid X_1, ..., X_{m-1}) = H(X_1, ..., X_m) - H(X_1, ..., X_{m-1}). \tag{2}$$

Then, based on the conditional entropy, the entropy rate of a random process is defined as:

$$\overline{H}(X) = \lim_{m \to \infty} H(X_m \mid X_1, ..., X_{m-1}). \tag{3}$$

The corrected conditional entropy is computed as a modification of Equation 3. First, the joint probabilities, $P(X_1 = x_1, ..., X_m = x_m)$ are replaced with empirically-derived probabilities. The data is binned into Q bins, i.e., values are converted to bin numbers from 1 to Q. The empirically-derived probabilities are then determined by the proportions of bin number sequences in the data. The entropy estimate and conditional entropy estimate, based on empirically-derived probabilities, are denoted as EN and CE respectively. Second, a corrective term, $perc(X_m) \cdot EN(X_1)$, is added to adjust for the limited number of sequences for increasing values of m [28]. The corrected conditional entropy, denoted as CCE, is computed as:

$$\begin{aligned} &CCE(X_m \mid X_1, ..., X_{m-1}) = \\ &\quad CE(X_m \mid X_1, ..., X_{m-1}) + perc(X_m) \cdot EN(X_1), \end{aligned} \tag{4}$$

where $perc(X_m)$ is the percentage of unique sequences of length m and $EN(X_1)$ is the entropy with m fixed at 1 or the first-order entropy.

The estimate of the entropy rate is the minimum of the corrected conditional entropy over different values of m. The minimum of the corrected conditional entropy is considered to be the best estimate of the entropy rate from the limited number of sequences.

4.2 Machine Learning Component

The machine learning component uses the content of tweets to detect spam. We have observed that most spam tweets are generated by bots and only very few of them are manually posted by humans. Thus, the presence of spam patterns usually indicates automation. Since tweets are text, determining if their content is spam can be reduced to a text classification problem. The text classification problem is formalized as $f : T \times C \rightarrow \{0, 1\}$, where f is the classifier, $T = \{t_1, t_2, ..., t_n\}$ are the texts to be classified, and $C = \{c_1, c_2, ..., c_k\}$ are the classes [31]. A value of 1 for $f(t_i, c_j)$ indicates that text t_i belongs to class c_j, whereas a value of 0 indicates it does not belong to that class. Bayesian classifiers are very effective in text classification, especially for email spam detection, so we employ Bayesian classification for our machine learning text classification component.

In Bayesian classification, deciding if a message belongs to a class, e.g., spam, is done by computing the corresponding probability based on its content, e.g., $P(C = spam|M)$, where M is a message and C is a class. If the probability is over a certain threshold, then the message is from that class.

The probability that a message M is spam, $P(spam|M)$, is computed from Bayes theorem:

$$P(spam|M) = \frac{P(M|spam)P(spam)}{P(M)} = \frac{P(M|spam)P(spam)}{P(M|spam)P(bot) + P(M|not\ spam)P(not\ spam)}. \quad (5)$$

The message M is represented as a feature vector $\langle f_1, f_2, ..., f_n \rangle$, where each feature f is one or more words in the message and each feature is assumed to be conditionally independent.

$$P(spam|M) = \frac{P(spam)\prod_{i=1}^{n} P(f_i|spam)}{P(spam)\prod_{i=1}^{n} P(f_i|spam) + P(not\ spam)\prod_{i=1}^{n} P(f_i|not\ spam)}. \quad (6)$$

The calculation of $P(spam|M)$ varies in different implementations of Bayesian classification. The implementation used for our machine learning component is CRM114 [4]. CRM114 is a powerful text classification system that offers a variety of different classifiers. The default classifier for CRM114 is Orthogonal Sparse Bigram (OSB), a variant of Bayesian classification, which has been shown to perform well for email spam filtering. OSB differs from other Bayesian classifiers in that it treats pairs of words as features.

4.3 Account Properties Component

Besides inter-tweet delay and tweet content, some Twitter account-related properties are very helpful for the user classification. As shown in Section 3.3, obvious difference exists between the human and bot categories. The first property is the URL ratio. The ratio indicates how often a user includes external URLs in its posted tweets. External URLs appear very often in tweets posted by a bot. Our measure shows, on average the ratio of bot is 97%, while that of human is much lower at 29%. Thus, a high ratio (e.g., close to one) suggests a bot and a low ratio implies a human.

The second property is tweeting device makeup. According to Table 1, about 70% tweets of human are posted via web and mobile devices (referred as manual devices), whereas about 87% tweets of bot are posted via API and other auto-piloted programs (referred as auto devices). The third property is the followers to friends ratio. Figure 3 clearly shows the difference between human and bot. The fourth property is link safety, i.e., to decide whether external links in tweets are malicious/phishing URLs or not. We use Google's Safe Browsing (GSB) API project [16], which allows us to check URLs against Google's constantly-updated blacklists of suspected phishing and malware pages. The component converts each URL[9] into hash values based on Google's rules, and performs the local lookup from the downloaded Google's blacklists. Appearance in Google's blacklists raises a red flag for security breach. GSB is also applied by Twitter for the link safety inspection [38]. The fifth property is whether a Twitter account is verified. No bot in our ground truth dataset is verified. The account verification suggests a human. The last property is the account registration date. According to Figure 5, 94.8% of bots were registered in 2009.

The account properties component extracts these properties from the user log, and sends them to the decision maker. It assists the entropy component and the machine-learning component to improve the classification accuracy.

4.4 Decision Maker

Given an unknown user, the decision maker uses the features identified by the above three components to determine whether it is a human, bot, or cyborg. It is built on Linear Discriminant Analysis (LDA) [26]. LDA is a statistical method to determine a linear combination of features that discriminate among multiple classes of samples. More specifically, its underlying idea is to determine whether classes differ in light of the means of a feature (or features), and then to use that feature (or features) to identify classes. It is very similar to analysis of variance (ANOVA) [29] and (logistic) regression analysis [19]. However, a big difference is that LDA has a fundamental assumption that independent variables are normally distributed. In other words, it is assumed that variables represent a sample from a multivariate normal distribution. Our classification involves three classes, human, bot and cyborg. Thus, it is a case of multiclass LDA. Multiclass LDA has the following key steps. First, it needs a training set and a test set that contain those samples already classified as one of the *C* classes. Samples in the two sets should not overlap with each other. Second, a discriminant model is created to use effective features to identify classes. Choosing features and assigning weights to features are the two important tasks in the model creation. In the early data collection stage, one usually includes several features to see which one(s) contributes to the discrimination. Some features are of very limited value for discrimination, and should be removed from the model. Our model uses *forward stepwise analysis*. In this way, the model is built step-by-step. At each step, all the features are evaluated, and the one that contributes the most to the discrimination is added into the model. The selection process continues to next step. Suppose m features, $< v_1, v_2, ..., v_m >$ are selected. Each class C_i has a classification function. With those functions, we can compute the classification score of an unknown sample for each class, by using the following linear equation:

$$S_i = w_{i0} + \sum_{i=1}^{n} w_{i1}v_1 + w_{i2} * v_2 + ... + w_{im} * v_m \quad (7)$$

where *i* denotes the respective class, S_i denotes the classification score of the sample for class C_i, w_{i0} denotes a constant for class C_i, and w_{ij} denotes the weight of *j*-th feature in class C_i.

The sample is classified into the class with the highest classification score. The model uses the training set to decide feature weights. Every sample in the training set is already known for the actual class it belongs to. The model keeps adjusting weights till it reaches the maximum accuracy for the training set. Third, the test set is used to validate the classification accuracy of the model. Since discriminant functions are derived from the training set, it is inappropriate to reuse it for the validation. The test set contains new data different from the training set, and generates more accurate validation results.

5. EVALUATION

In this section, we first evaluate the accuracy of our classification system based on the ground truth set that includes both the train-

[9]For a shortened URL, our component uses PHP cURL to get the original one from the redirected HTTP response header, instead of actually visiting the page.

Table 2: Multi-class LDA Weights

	Human	Cyborg	Bot
Constant	-25.9879	-15.7787	-17.2416
Entropy	14.2524	9.7128	4.4136
Bayesian text	-0.0018	0.0164	0.1366
URL ratio	-3.4474	3.3059	8.5222
Manual device %	16.4601	13.0164	13.0950
Auto device %	8.5910	7.6849	18.3765
Followers to friends ratio	0.0007	0.0002	0.0003

ing and test datasets. Then, we apply the system to classify the entire dataset of over 500,000 users collected. With the classification results, we further speculate the current composition of Twitter user population. Finally, we discuss the robustness of the proposed classification system against possible evasions.

5.1 Methodology

As shown in Figure 8, the components of the classification system collaborate in the following way. The entropy component calculates the entropy (and corrected conditional entropy) of inter-tweet delays of a Twitter user. The entropy component only processes logs with more than 100 tweets[10]. This limit helps reduce noise in detecting automation. A lower entropy indicates periodic or regular timing of tweeting behavior, a sign of automation, whereas a higher entropy implies irregular behavior, a sign of human participation. The machine learning component determines if the tweet content is either spam or not, based on the text patterns it has learned. The content feature value is set to -1 for spam but 1 for non-spam. The account properties component checks all the properties mentioned in Section 4.3, and generates a real-number-type value for each property. Given a Twitter user, the above three components generate a set of features and input them into the decision maker. For each class, namely human, bot and cyborg, the decision maker computes a classification score for the user, and classifies it into the class with the highest score. The training of the classification system and its accuracy are detailed as follows.

5.2 Classification System Training

The classification system needs to be trained before being used. In particular, the machine learning component and the decision maker require training. The machine learning component is trained on spam and non-spam datasets. The spam dataset consists of spam tweets and spam external URLs, which are detected during the creation of the ground truth set. Some advanced spam bots intentionally inject non-spam tweets (usually in the format of pure text without URLs, such as adages[11]) to confuse human users. Thus, we do not include such vague tweets without external URLs. The non-spam dataset consists of all human tweets and cyborg tweets without external URLs. Most human tweets do not carry spam. Cyborg tweets with links are hard to determine without checking linked web pages. They can be either spam or non-spam. Thus, we do not include this type of tweets in either dataset. Training the component with up-to-date spam text patterns on Twitter helps improve the accuracy.

The decision maker is trained to determine the weights of the different features for classification. We use Statistica, a statistical tool [33], to calculate the feature weights. More specifically, the datasheet of feature values and the actual class of users in the training set are inputted into the classifier. LDA generates a weight table (Table 2) to achieve the maximum accuracy. In other words, it includes as many users as possible, whose classified class matches actual class. The weights are then used by the decision maker to classify users.

The larger the (standardized) weight, the larger is the unique contribution of the corresponding feature to the discrimination. Table 2 shows that, entropy, URL ratio, and manual/auto device percentage are the important features for the classifier. Only those shown to be statistically significant should be used for classification, and non-significant ones should be ignored. Thus, some features collected by the account properties component in Section 4.3, including followers to friends ratio, link safety, account verification and registration date, are excluded from the classifier.

Here we briefly explain why several features, such as followers to friends ratio, link safety, account verification, and registration date, are not as important in the actual discrimination as expected. Bots used to have more friends than followers [25], and the ratio is less than one in this situation. However, there have emerged some more sophisticated bots that unfollow their friends if they do not follow back within a certain amount of time. They cunningly keep the ratio close to one. This strategy makes the ratio feature less useful. Most spam bots spread spam links on Twitter, instead of phishing or malicious links which are the primary target of the link safety inspector. Only 0.2% of the users in the training set do not pass the link safety inspection. Thus, the link safety feature has little weight under LDA due to its statistical insignificance. Similarly, account verification has a very small weight, because it is also quite rare. Only 1.8% of the users are verified. Lastly, account registration dates greatly overlap among bots, humans, and cyborgs, making this feature not useful for discrimination as well.

5.3 Classification System Accuracy

To validate the accuracy of our proposed classification system, we create a test set containing one thousand users of each class. It does not share any samples with the training set. The confusion matrix listed in Table 3 shows the classification results on the test set.

The "Actual" rows in Table 3 denote the actual classes of the users, and the "Classified" columns denote the classes of the users as decided by the classification system. For example, 949 in the "Human" row and column means that 949 humans are classified (correctly) as humans, whereas 51 in the "Human" row and "Cyborg" column means that 51 humans are classified (incorrectly) as cyborgs. There is no misclassification between human and bot.

We examine the logs of those users being classified by mistake, and analyze each category as follows.

- For the human category, 5.1% of human users are classified as cyborg by mistake. One reason is that, the overall scores of some human users are lowered by spam content penalty. The tweet size is up to 140 characters. Some patterns and phrases are used by both human and bot, such as "I post my online marketing experience at my blog at http://bit.ly/xT6klM. Please ReTweet it." Another reason is that the tweeting interval distribution of some human users is slightly lower than the entropy means, and they are penalized for that.

- For the bot category, 6.3% of bots are wrongly categorized as cyborg. The main reason is that, most of them escape the spam penalty from the machine learning component. Some spam tweets use very obscure text content, like "you should check it out since it's really awesome. http://bit.ly/xT6klM". Without checking the spam link, the component cannot determine if the tweet is spam merely based on the text.

- For the cyborg category, 9.8% of cyborgs are mis-classified as human, and 7.4% of them are mis-classified as bot. A cyborg can be either a human-assisted bot or a bot-assisted human. A strict policy could categorize cyborg as bot, while a loose one may categorize it as human.

Overall, our classification system can accurately differentiate human from bot. However, it is much more challenging for a classification system to distinguish cyborg from human or bot.

[10]The inter-tweet span could be wild on Twitter. An account may be inactive for months, but suddenly tweets at an intensive frequency for a short-term, and then enters hibernation again. It generates noise to the entropy component. Thus, the entropy component does not process logs with less than 100 tweets. Besides, in practice it is nearly impossible to determine automation based on a very limited number of tweets.

[11]A typical content pattern is listed as follows. Tweet 1, A friend in need is a friend in deed. Tweet 2, Danger is next neighbor to security. Tweet 3, Work home and make $3k per month. Check out how, http://tinyurl.com/bF234T.

Table 3: Confusion Matrix

		Classified				
		Human	Cyborg	Bot	Total	True Pos.%
Actual	Human	949	51	0	1000	94.90%
	Cyborg	98	828	74	1000	82.80%
	Bot	0	63	937	1000	93.70%

5.4 Twitter Composition

We further use the classification system to automatically classify our whole dataset of over 500,000 users. We can speculate the current composition of Twitter user population based on the classification results. The system classifies 48.7% of the users as human, 37.5% as cyborg, and 13.8% as bot. Thus, we speculate the population proportion of human, cyborg and bot category roughly as 5:4:1 on Twitter.

5.5 Resistance to Evasion

Now we discuss the resistance of the classification system to possible evasion attempts made by bots. Bots may deceive certain features, such as the followers to friends ratio as mentioned before. However, our system has two critical features that are very hard for bots to evade. The first feature is tweeting device makeup, which corresponds to the manual/auto device percentage in Table 2. Manual device refers to web and mobile devices, while auto device refers to API and other auto-piloted programs (see Section 4.3). Tweeting via web requires a user to login and manually post via the Twitter website in a browser. Posting via HTTP form is considered by Twitter as API. Furthermore, currently it is impractical or expensive to run a bot on a mobile device to frequently tweet. As long as Twitter can correctly identify different tweeting platforms, device makeup is an effective metric for bot detection. The second feature is URL ratio. Considering the limited tweet length that is up to 140 characters, most bots have to include a URL to redirect users to external sites. Thus, a high URL ratio is another effective metric for bot detection. Other features like timing entropy, bot could mimic human behaviors but at the cost of much reduced tweeting frequency. We will continue to explore new features emerging with the Twitter development for more effective bot detection in the future.

6. CONCLUSION

In this paper, we have studied the problem of automation by bots and cyborgs on Twitter. As a popular web application, Twitter has become a unique platform for information sharing with a large user base. However, its popularity and very open nature have made Twitter a very tempting target for exploitation by automated programs, i.e., bots. The problem of bots on Twitter is further complicated by the key role that automation plays in everyday Twitter usage.

To better understand the role of automation on Twitter, we have measured and characterized the behaviors of humans, bots, and cyborgs on Twitter. By crawling Twitter, we have collected one-month of data with over 500,000 Twitter users with more than 40 million tweets. Based on the data, we have identified features that can differentiate humans, bots, and cyborgs on Twitter. Using entropy measures, we have determined that humans have complex timing behavior, i.e., high entropy, whereas bots and cyborgs are often given away by their regular or periodic timing, i.e., low entropy. In examining the text of tweets, we have observed that a high proportion of bot tweets contain spam content. Lastly, we have discovered that certain account properties, like external URL ratio and tweeting device makeup, are very helpful on detecting automation.

Based on our measurements and characterization, we have designed an automated classification system that consists of four main parts: the entropy component, the machine learning component, the account properties component, and the decision maker. The entropy component checks for periodic or regular tweet timing patterns; the machine learning component checks for spam content; and the account properties component checks for abnormal values of Twitter-account-related properties. The decision maker summarizes the identified features and decides whether the user is a human, bot, or cyborg. The effectiveness of the classification system is evaluated through the test dataset. Moreover, we have applied the system to classify the entire dataset of over 500,000 users collected, and speculated the current composition of Twitter user population based on the classification results.

7. REFERENCES

[1] Amazon comes to twitter. `http://www.readwriteweb.com/archives/amazon_comes_to_twitter.php` [Accessed: Dec. 20, 2009].

[2] Barack obama uses twitter in 2008 presidential campaign. `http://twitter.com/BarackObama/` [Accessed: Dec. 20, 2009].

[3] Best buy goes all twitter crazy with @twelpforce. `http://twitter.com/in_social_media/status/2756927865` [Accessed: Dec. 20, 2009].

[4] The crm114 discriminator. `http://crm114.sourceforge.net/` [Accessed: Sept. 12, 2009].

[5] Alexa. The top 500 sites on the web by alexa. `http://www.alexa.com/topsites` [Accessed: Jan. 15, 2010].

[6] Meeyoung Cha, Haewoon Kwak, Pablo Rodriguez, Yong-Yeol Ahn, and Sue Moon. I tube, you tube, everybody tubes: analyzing the world's largest user generated content video system. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, San Diego, CA, USA, 2007.

[7] Meeyoung Cha, Alan Mislove, and Krishna P. Gummadi. A measurement-driven analysis of information propagation in the flickr social network. In *Proceedings of the 18th International Conference on World Wide Web*, Madrid, Spain, 2009.

[8] Thomas M. Cover and Joy A. Thomas. *Elements of information theory*. Wiley-Interscience, New York, NY, USA, 2006.

[9] Marcel Dischinger, Andreas Haeberlen, Krishna P. Gummadi, and Stefan Saroiu. Characterizing residential broadband networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet Measurement*, San Diego, CA, USA, 2007.

[10] Il-Chul Moon Dongwoo Kim, Yohan Jo and Alice Oh. Analysis of twitter lists as a potential source for discovering latent characteristics of users. In *To appear on CHI 2010 Workshop on Microblogging: What and How Can We Learn From It?*, 2010.

[11] Henry J. Fowler and Will E. Leland. Local area network traffic characteristics, with implications for broadband network congestion management. *IEEE Journal of Selected Areas in Communications*, 9(7), 1991.

[12] Steven Gianvecchio and Haining Wang. Detecting covert timing channels: An entropy-based approach. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security*, Alexandria, VA, USA, October-November 2007.

[13] Steven Gianvecchio, Zhenyu Wu, Mengjun Xie, and Haining Wang. Battle of botcraft: fighting bots in online games with human observational proofs. In *Proceedings of the 16th ACM conference on Computer and Communications Security*, Chicago, IL, USA, 2009.

[14] Steven Gianvecchio, Mengjun Xie, Zhenyu Wu, and Haining Wang. Measurement and classification of humans and bots in internet chat. In *Proceedings of the 17th USENIX Security symposium*, San Jose, CA, 2008.

[15] Minas Gjoka, Maciej Kurant, Carter T Butts, and Athina Markopoulou. Walking in facebook: A case study of unbiased sampling of osns. In *Proceedings of the 27th IEEE International Conference on Computer Communications*, San Diego, CA, USA, March 2010.

[16] Google. Google safe browsing API. `http://code.google.com/apis/safebrowsing/`

[Accessed: Feb. 5, 2010].

[17] Paul Graham. A plan for spam, 2002. http://www.paulgraham.com/spam.html [Accessed: Jan. 25, 2008].

[18] Monika R. Henzinger, Allan Heydon, Michael Mitzenmacher, and Marc Najork. On near-uniform url sampling. In *Proceedings of the 9th International World Wide Web Conference on Computer Networks*, Amsterdam, The Netherlands, May 2000.

[19] Christopher M. Hill and Linda C. Malone. Using simulated data in support of research on regression analysis. In *WSC '04: Proceedings of the 36th conference on Winter simulation*, 2004.

[20] B A Huberman and T Hogg. Complexity and adaptation. *Phys. D*, 2(1-3), 1986.

[21] A. L. Hughes and L. Palen. Twitter adoption and use in mass convergence and emergency events. In *Proceedings of the 6th International ISCRAM Conference*, Gothenburg, Sweden, May 2009.

[22] H. Husna, S. Phithakkitnukoon, and R. Dantu. Traffic shaping of spam botnets. In *Proceedings of the 5th IEEE Conference on Consumer Communications and Networking*, Las Vegas, NV, USA, January 2008.

[23] Bernard J. Jansen, Mimi Zhang, Kate Sobel, and Abdur Chowdury. Twitter power: Tweets as electronic word of mouth. *American Society for Information Science and Technology*, 60(11), 2009.

[24] Akshay Java, Xiaodan Song, Tim Finin, and Belle Tseng. Why we twitter: understanding microblogging usage and communities. In *Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 Workshop on Web Mining and Social Network Analysis*, San Jose, CA, USA, 2007.

[25] Balachander Krishnamurthy, Phillipa Gill, and Martin Arlitt. A few chirps about twitter. In *Proceedings of the First Workshop on Online Social Networks*, Seattle, WA, USA, 2008.

[26] G. J. McLachlan. *Discriminant Analysis and Statistical Pattern Recognition*. Wiley Interscience, 2004.

[27] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, San Diego, CA, USA, 2007.

[28] A Porta, G Baselli, D Liberati, N Montano, C Cogliati, T Gnecchi-Ruscone, A Malliani, and S Cerutti. Measuring regularity by means of a corrected conditional entropy in sympathetic outflow. *Biological Cybernetics*, Vol. 78(No. 1), January 1998.

[29] P. Real. A generalized analysis of variance program utilizing binary logic. In *ACM '59: Preprints of papers presented at the 14th national meeting of the Association for Computing Machinery*, New York, NY, USA, 1959.

[30] Erick Schonfeld. Costolo: Twitter now has 190 million users tweeting 65 million times a day. http://techcrunch.com/2010/06/08/twitter-190-million-users/ [Accessed: Sept. 26, 2010].

[31] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM Computing Surveys*, Vol. 34(No. 1), 2002.

[32] Kate Starbird, Leysia Palen, Amanda Hughes, and Sarah Vieweg. Chatter on the red: What hazards threat reveals about the social life of microblogged information. In *Proceedings of the ACM 2010 Conference on Computer Supported Cooperative Work*, February 2010.

[33] Statsoft. Statistica, a statistics and analytics software package developed by statsoft. http://www.statsoft.com/support/download/brochures/ [Accessed: Mar. 12, 2010].

[34] Brett Stone-Gross, Marco Cova, Lorenzo Cavallaro, Bob Gilbert, Martin Szydlowski, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna. Your botnet is my botnet: analysis of a botnet takeover. In *Proceedings of the 16th ACM conference on Computer and Communications Security*, Chicago, IL, USA, 2009.

[35] J. Sutton, Leysia Palen, and Irina Shlovski. Back-channels on the front lines: Emerging use of social media in the 2007 southern california wildfires. In *Proceedings of the 2008 ISCRAM Conference*, Washington, DC, USA, May 2008.

[36] Alan M. Turing. Computing machinery and intelligence. *Mind*, Vol. 59:433–460, 1950.

[37] Tweetadder. Automatic twitter software. http://www.tweetadder.com/ [Accessed: Feb. 5, 2010].

[38] Twitter. How to report spam on twitter. http://help.twitter.com/entries/64986 [Accessed: May. 30, 2010].

[39] Twitter. Twitter api wiki. http://apiwiki.twitter.com/ [Accessed: Feb. 5, 2010].

[40] Mengjun Xie, Zhenyu Wu, and Haining Wang. Honeyim: Fast detection and suppression of instant messaging malware in enterprise-like networks,. In *Proceedings of the 23rd Annual Computer Security Applications Conference*, Miami Beach, FL, USA, 2007.

[41] Mengjun Xie, Heng Yin, and Haining Wang. An effective defense against email spam laundering. In *Proceedings of the 13th ACM conference on Computer and Communications Security*, Alexandria, VA, USA, 2006.

[42] Jeff Yan. Bot, cyborg and automated turing test. In *Proceedings of the 14th International Workshop on Security Protocols*, Cambridge, UK, March 2006.

[43] Sarita Yardi, Daniel Romero, Grant Schoenebeck, and Danah Boyd. Detecting spam in a twitter network. *First Monday*, 15(1), January 2010.

[44] Jonathan A. Zdziarski. *Ending Spam: Bayesian Content Filtering and the Art of Statistical Language Classification*. No Starch Press, 2005.

[45] Dejin Zhao and Mary Beth Rosson. How and why people twitter: the role that micro-blogging plays in informal communication at work. In *Proceedings of the ACM 2009 International Conference on Supporting Group Work*, Sanibel Island, FL, USA, 2009.

Cujo: Efficient Detection and Prevention of Drive-by-Download Attacks

Konrad Rieck
Machine Learning Group
Technische Universität Berlin, Germany
konrad.rieck@tu-berlin.de

Tammo Krueger
Intelligent Data Analysis
Fraunhofer Institute FIRST, Germany
tammo.krueger@tu-berlin.de

Andreas Dewald
Laboratory for Dependable Distributed Systems
University of Mannheim, Germany
andreas.dewald@uni-mannheim.de

ABSTRACT

The JavaScript language is a core component of active and dynamic web content in the Internet today. Besides its great success in enhancing web applications, however, JavaScript provides the basis for so-called *drive-by downloads*—attacks exploiting vulnerabilities in web browsers and their extensions for unnoticeably downloading malicious software. Due to the diversity and frequent use of obfuscation in these attacks, static code analysis is largely ineffective in practice. While dynamic analysis and honeypots provide means to identify drive-by-download attacks, current approaches induce a significant overhead which renders immediate prevention of attacks intractable.

In this paper, we present Cujo, a system for automatic detection and prevention of drive-by-download attacks. Embedded in a web proxy, Cujo transparently inspects web pages and blocks delivery of malicious JavaScript code. Static and dynamic code features are extracted on-the-fly and analysed for malicious patterns using efficient techniques of machine learning. We demonstrate the efficacy of Cujo in different experiments, where it detects 94% of the drive-by downloads with few false alarms and a median run-time of 500 ms per web page—a quality that, to the best of our knowledge, has not been attained in previous work on detection of drive-by-download attacks.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; I.5.1 [**Pattern Recognition**]: Models—*Statistical*

Keywords

Drive-by downloads, web security, static code analysis, dynamic code analysis, machine learning

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

1. INTRODUCTION

The JavaScript language is a ubiquitous tool for providing active and dynamic content in the Internet. The vast majority of web sites, including large social networks, such as Facebook and Twitter, makes heavy use of JavaScript for enhancing the appearance and functionality of their services. In contrast to server-based scripting languages, JavaScript code is executed in the web browser of the client and thus provides means for directly interacting with the user and the browser environment. Although the execution of JavaScript code at the client is restricted by several security policies, the interaction with the browser and its extensions alone gives rise to a severe security threat.

JavaScript is increasingly used as basis for *drive-by downloads*, attacks exploiting vulnerabilities in web browsers and their extensions for unnoticeably downloading malicious software [see 15, 16]. These attacks take advantage of the tight integration of JavaScript with the browser environment to exploit different types of vulnerabilities and eventually assume control of the web client. Due to the complexity of browsers and their extensions, there exist numerous of these vulnerabilities, ranging from insecure interfaces of third-party extensions to buffer overflows and memory corruptions [5, 7, 11]. Four of the top five most attacked vulnerabilities observed by Symantec in 2009 have been such client-side vulnerabilities involved in drive-by-download attacks [2].

As a consequence, detection of drive-by downloads has gained a focus in security research. Two classes of defense measures have been proposed to counteract this threat: First, several security vendors have equipped their products with rules and heuristics for identifying malicious code directly at the client. This static code analysis, however, is largely obstructed by the frequent use of obfuscation in drive-by downloads. A second strain of research has thus studied detection of drive-by downloads using dynamic analysis, for example using code emulation [8, 17], sandboxing [4, 6, 16] and client honeypots [14, 16, 21]. Although effective in detecting attacks, these approaches suffer from either of two shortcomings: Some approaches are limited to specific attack types, such as heap spraying [e.g., 8, 17], whereas the more general approaches [e.g., 4, 14] induce an overhead prohibitive for preventing attacks at the client.

As a remedy, we present Cujo[1], a system for detection and prevention of drive-by-download attacks, which combines advantages of static and dynamic analysis concepts.

[1] Cujo = "Classification of Unknown Javascript cOde"

Embedded in a web proxy, Cujo transparently inspects web pages and blocks delivery of malicious JavaScript code to the client. The analysis and detection methodology implemented in this system rests on the following contributions of this paper:

- *Lightweight JavaScript analysis.* We devise efficient methods for static and dynamic analysis of JavaScript code, which provide expressive analysis reports with very small run-time overhead.
- *Generic feature extraction.* For the generic detection of drive-by downloads, we introduce a mapping from analysis reports to a vector space that is spanned by short analysis patterns and independent of specific attack characteristics.
- *Learning-based detection.* We apply techniques of machine learning for generating detection models for static and dynamic analysis, which spares us from manually crafting and updating detection rules as in current security products.

An empirical evaluation with 200,000 web pages and 600 real drive-by-download attacks demonstrates the efficacy of this approach: Cujo detects 94% of the attacks with a false-positive rate of 0.002%, corresponding to 2 false alarms in 100,000 visited web sites, and thus is almost on par with offline analysis systems, such as Jsand [4]. In terms of run-time, however, Cujo significantly surpasses these systems. With caching enabled, Cujo provides a median run-time of 500 ms per web page, including downloading of web page content and full analysis of JavaScript code. To the best of our knowledge, Cujo is the first system capable of effectively and efficiently blocking drive-by downloads in practice.

The rest of this paper is organized as follows: Cujo and its detection methodology are introduced in Section 2 including JavaScript analysis, feature extraction and learning-based detection. Experiments and comparisons to related techniques are presented in Section 3. Related work is discussed in Section 4 and Section 5 concludes.

2. METHODOLOGY

Drive-by-download attacks can take almost arbitrary structure and form, depending on the exploited vulnerabilities as well as the use of obfuscation. Efficient analysis and detection of these attacks is a challenging problem, which requires careful balancing of detection and run-time performance. We address this problem by applying lightweight static and dynamic code analysis, thereby providing two complementary views on JavaScript code. To avoid manually crafting detection rules for each of these views, we employ techniques of machine learning, which enable generalizing from known attacks and allow to automatically construct detection models. A schematic view of the resulting system is presented in Figure 1.

Cujo is embedded in a web proxy and transparently inspects the communication between a web client and a web service. Prior to delivery of web page data from the service to the client, Cujo performs a series of analysis steps and depending on their results blocks pages likely containing malicious JavaScript code. To improve processing performance, two analysis caches are employed: First, all incoming web data is cached to reduce loading times and, second, analysis results are cached, if all embedded and external code associated with a web page has not changed within a limited period of time.

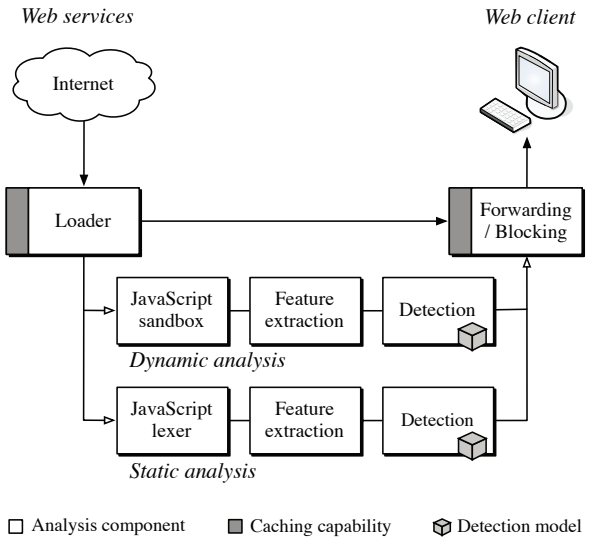


Figure 1: Schematic depiction of Cujo.

2.1 JavaScript Analysis

As first analysis step, we aim at efficiently getting a comprehensive view on JavaScript code. To this end, we inspect all HTML and XML documents passing our system for occurrences of JavaScript. For each requested document, we extract all code blocks embedded using the HTML tag `script` and contained in HTML event handlers, such as `onload` and `onmouseover`. Moreover, we recursively pre-load all external code referenced in the document, including scripts, frames and iframes, to obtain the complete code base of the web page. All code blocks of a requested document are then merged for further static and dynamic analysis.

As an example used throughout the following sections, we consider the JavaScript code shown in Figure 2. The code is obfuscated using a simple substitution cipher and contains a routine for constructing a NOP sled, an array of NOP instructions common in most memory corruption attacks. Analysis reports for the static and dynamic analysis of this code snippet are shown in Figure 3 and 4, respectively.

```
1 a = "";
2 b = "{@xqhvfdsh+%(x<3<3%,>zkloh+{1ohqjwk?4333,{.@{>";
3 for (i = 0; i < b.length; i++) {
4     c = b.charCodeAt(i) - 3;
5     a += String.fromCharCode(c);
6 }
7 eval(a);
```

Figure 2: Obfuscated JavaScript code for generating a NOP sled.

2.1.1 Static Analysis

Our static analysis relies on basic principles of compiler design [3]: Before the source code of a program can be interpreted or compiled, it needs to be decomposed into lexical tokens, which are then fed to the actual parser. The static

analysis component in CUJO takes advantage of this process and efficiently extracts lexical tokens from the JavaScript code of a web page using a customized YACC grammar.

The lexical analysis closely follows the language specification of JavaScript [1], where source code is sequentially decomposed into keywords, punctuators, identifiers and literals. As the actual names of identifiers do not contribute to the structure of code, we replace them by the generic token `ID`. Similarly, we encode numerical literals by `NUM` and string literals by `STR`. An example of this basic decomposition is illustrated in the following

```
x = foo(y) + "bar";  ⟶  ID = ID ( ID ) + STR ;
```

where keywords and punctuators are represented by individual tokens, while identifiers and strings are subsumed by the generic tokens `ID` and `STR`, respectively.

To further strengthen our static analysis for detection of drive-by-download attacks, we make two refinements to the lexical analysis. First, we additionally encode the length of string literals as decimal logarithm. That is, `STR.01` refers to a string with up to 10^1 characters, `STR.02` to a string with up to 10^2 characters and so on. Second, we add `EVAL` as a new keyword to the analysis. Both refinements target common constructs of drive-by-download attacks, which involve string operations and calls to the `eval()` function.

Although obfuscation techniques may hide code from this static analysis, several programming constructs and structures can be distinguished in terms of lexical tokens. As an example, Figure 3 shows an analysis report of lexical tokens for the example code given in Figure 2. While the actual code for generating a NOP sled is hidden in the encrypted string (line 2), several patterns indicative for obfuscation, such as the decryption loop (line 3–5) and the call to `EVAL` (line 7), are accessible to means of detection techniques

2.1.2 Dynamic Analysis

For dynamic analysis, we adopt an enhanced version of ADSANDBOX, a lightweight JavaScript sandbox developed by Dewald et al. [6]. The sandbox takes the code associated with a web page and executes it within the JavaScript interpreter SPIDERMONKEY[2]. The interpreter operates in a virtual browser environment and reports all operations changing the state of this environment. Additionally, we invoke all event handlers of the code to trigger functionality dependent on external events. As result of this dynamic analysis, the sandbox provides a report containing all monitored operations of a given JavaScript code.

To emphasize behavior related to drive-by-download attacks, we extend the dynamic code analysis with *abstract operations*, which represent patterns of common attack activity. These abstract operations are encoded as regular expressions and matched on-the-fly during the monitoring of JavaScript code. Currently, CUJO supports two of these operations: First, we indicate typical behavior of heap-spraying attacks, such as excessive allocation of memory chunks by appending the operation `HEAP SPRAYING` and, second, we mark the use of browser functions inducing a re-evaluation of strings by the interpreter using the operation `PSEUDO-EVAL`. While both abstract operations are indicative for particular attacks, they are not sufficient for detection alone and a full inspection of behavior reports is required.

[2]SpiderMonkey, `http://www.mozilla.org/js/SpiderMonkey`

```
1 ID = STR.00 ;
2 ID = STR.02 ;
3 FOR ( ID = NUM ; ID < ID . ID ; ID ++ ) {
4     ID = ID . ID ( ID ) - NUM ;
5     ID + = ID . ID ( ID ) ;
6 }
7 EVAL ( ID ) ;
```

Figure 3: Example of static analysis.

```
1  SET  global.a TO ""
2  SET  global.b TO "{@xqhvfdsh+%(x<3<3%,>zkloh
3                   +{1ohqjwk?4333,{.@{>"
4  SET  global.i TO "0"
5  CALL charCodeAt
6  SET  global.c TO "120"
7  CALL fromCharCode
8  SET  global.a TO "x"
9  ...
10 SET  global.a TO "x=unescape("%u9090");
11                  while(x.length<1000)x+=x;"
12 SET  global.i TO "46"
13 CALL eval
14 CALL unescape
15 SET  global.x TO "<90><90>"
16 SET  global.x TO "<90><90><90><90>"
17 ...
18 SET  global.x TO "<90> ... 1024 bytes ... <90>"
```

Figure 4: Example of dynamic analysis.

Although this lightweight analysis provides only a coarse view on the behavior of JavaScript code in comparison to offline analysis [e.g., 4, 14, 21], it enables accurate detection of drive-by downloads with a median run-time of less than 400 ms per web page, as demonstrated in Section 3.4. As an example, Figure 4 shows a behavior report for the code snippet given in Figure 2. The first lines of the report cover the decryption of the obfuscated string, which is finally revealed in lines 10–11. Starting with the call to `eval`, this string is evaluated by the interpreter and results in the construction of a NOP sled with 1024 bytes in line 18.

2.2 Feature Extraction

In the second analysis step, we extract features from the analysis reports of static and dynamic analysis, suitable for application of detection methods. In contrast to previous work, we propose a generic feature extraction, which is independent of particular attack characteristics and allows to jointly process reports of static and dynamic analysis.

2.2.1 Q-gram Features

Our feature extraction builds on the concept of q-grams, which has been widely studied in the field of intrusion detection [e.g., 10, 18, 22]. To unify the representation of static and dynamic analysis, we first partition each report into a sequence of words using white-space characters. We then move a fixed-length window over each report and extract subsequences of q words at each position, so-called q-grams. The following example shows the extraction of q-grams with $q = 3$ for two code snippets of static and dynamic analysis, respectively,

$$\texttt{ID = ID + NUM} \longrightarrow \big\{ (\texttt{ID = ID}), (\texttt{= ID +}), (\texttt{ID + NUM}) \big\},$$

$$\texttt{SET a.b to "x"} \longrightarrow \big\{ (\texttt{SET a.b to}), (\texttt{a.b to "x"}) \big\}.$$

As a result of this extraction, each report is represented by a set of q-grams, which reflect short patterns and provide the basis for mapping analysis reports to a vector space.

Intuitively, we are interested in constructing a vector space, where analysis reports sharing several q-grams lie close to each other, while reports with dissimilar content are separated by large distances. To establish such a mapping, we associate each q-gram with one particular dimension in the vector space. Formally, this vector space is defined using the set S of all possible q-grams, where a corresponding mapping function for a report x is given by

$$\phi : x \to \big(\phi_s(x)\big)_{s \in S}$$

$$\text{with } \phi_s(x) = \begin{cases} 1 & \text{if } x \text{ contains the } q\text{-gram } s, \\ 0 & \text{otherwise.} \end{cases}$$

The function ϕ maps a report x to the vector space $\mathbb{R}^{|S|}$ such that all dimensions associated with q-grams contained in x are set to one and all other dimensions are zero. To avoid an implicit bias on the length of reports, we normalize $\phi(x)$ to one, that is, we set $||\phi(x)|| = 1$. As a result of this normalization, a q-gram counts more in a report that has fewer distinct q-grams. That is, changing a constant amount of tokens in a report containing repetitive structure has more impact on the vector than in an analysis report comprising several different patterns.

2.2.2 Efficient Q-gram Representation

At the first glance, the mapping ϕ seems inappropriate for efficient analysis: the set S covers all possible q-grams of words and induces a vector space of very large dimension. Fortunately, the number of q-grams contained in a report is linear in its length. An analysis report x containing m words comprises at most $(m - q)$ different q-grams. Consequently, only $(m - q)$ dimensions are non-zero in the vector $\phi(x)$, irrespective of the dimension of the vector space. It thus suffices to only store the q-grams contained in each report x for a sparse representation of the vector $\phi(x)$, for example, using efficient data structures such as sorted arrays [19] or Bloom filters [22]. As demonstrated in Section 3.4, this sparse representation of feature vectors provides the basis for very efficient feature extraction with median run-times below 1 ms per analysis report.

2.3 Learning-based Detection

As final analysis step of Cujo, we present a learning-based detection of drive-by-download attacks, which builds on the vectorial representation of analysis reports. The application of machine learning spares us from manually constructing and updating detection rules for static and dynamic code analysis, and thereby limits the delay to detection of novel drive-by downloads.

2.3.1 Support Vector Machines

For automatically generating detection models from the reports of attacks and benign JavaScript code, we apply the technique of *Support Vector Machines* (SVM) [see 13, 20]. Given vectors of two classes as training data, an SVM determines a hyperplane that separates both classes with maximum margin. In our setting, one of these classes is associated with analysis reports of drive-by downloads, whereas the other class corresponds to reports of benign web pages. An unknown report $\phi(x)$ is now classified by mapping it to the vector space and checking if it falls on either the malicious or benign side of the hyperplane. This learning-based detection of drive-by downloads is illustrated in Figure 5.

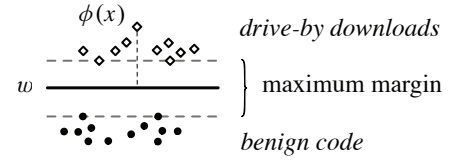


Figure 5: Schematic vector representation of analysis reports with maximum-margin hyperplane.

Formally, the detection model of an SVM corresponds to a vector w and bias b, specifying the direction and offset of the hyperplane in the vector space. The corresponding detection function f is given by

$$f(x) = \langle \phi(x), w \rangle + b = \sum_{s \in S} \phi_s(x) \cdot w_s + b.$$

and returns the orientation of $\phi(x)$ with respect to the hyperplane. That is, $f(x) > 0$ indicates malicious activity in the report x and $f(x) \leq 0$ corresponds to benign data.

In contrast to many other learning techniques, SVMs possess the ability to compensate a certain amount of noise in the labels of the training data—a crucial property for practical application of Cujo. This ability renders the learning process robust to a minor amount of unknown attacks in the benign portion of the training data and enables generating accurate detection models, even if some of the web pages labeled as benign data contain drive-by-download attacks. Theory and further details on this ability of SVMs are discussed in [13, 20].

2.3.2 Efficient Classification of Q-grams

For efficiently computing f, we again exploit the sparse representation of vectors induced by ϕ. Given a report x, we know that only q-grams contained in x have non-zero entries in $\phi(x)$, that is, all other dimensions in $\phi(x)$ are zero and do not contribute to the computation of $f(x)$. Hence, we can simplify the detection function f as follows

$$f(x) = \sum_{s \in S} \phi_s(x) \cdot w_s + b = \sum_{s \text{ in } x} \phi_s(x) \cdot w_s + b,$$

where we determine $f(x)$ by simply looking up the values w_s for each q-gram contained in x. As a consequence, the classification of a report can be carried out with linear time complexity and a median run-time below 0.2 ms per report (cf. Section 3.4). For learning the detection model of the SVM we employ LibLinear [9], a fast SVM library which enables us to train detection models from 100,000 reports in 120 seconds for dynamic analysis and in 50 seconds for static analysis.

2.3.3 Explanation

In practice, a detection systems must not only flag malicious events but also provide insights into the detection process, such that attack patterns and exploited vulnerabilities can be inspected during operation. Fortunately, we can adapt the detection function for explaining the decision process of the SVM. During computation of f, we additionally store the individual contribution $\phi_s(x) \cdot w_s$ of each q-gram to

the final detection score $f(x)$. If an explanation is requested, we output the q-grams with largest contribution and thereby present those analysis patterns that shifted the analysis report x to the positive side of the hyperplane. We illustrate this concept in Section 3.3, where we present explanations for detections of drive-by-download attacks using reports of static and dynamic analysis.

The learning-based detection completes the design of our system Cujo. As illustrated in Figure 1, Cujo uses two independent processing chains for static and dynamic code analysis, where an alert is reported if one of the detection models indicates a drive-by download.

This combined detection renders evasion of our system difficult, as it requires the attacker to cloak his attacks from both, static and dynamic analysis. While static analysis alone can be thwarted through massive obfuscation, the hidden code needs to be decrypted during run-time which in turn can be tracked by dynamic analysis. Similarly, if fewer obfuscation is used and the attacker tries to spoil the sandbox emulation, patterns of the respective code might be visible to static analysis. Although this argumentation does not rule out evasion in general, it clearly shows the effort necessary for evading our system.

3. EVALUATION

After presenting the detection methodology of Cujo, we turn to an empirical evaluation of its performance. In particular, we conduct experiments to study the detection and run-time performance in detail. Before presenting these experiments, we introduce our data sets of drive-by-download attacks and benign web pages.

3.1 Data Sets

We consider two data sets containing URLs of benign web pages, *Alexa-200k* and *Surfing*, which are listed in Table 1(a). The *Alexa-200k* data set corresponds to the 200,000 most visited web pages in the Internet as listed by Alexa[3] and covers a wide range of JavaScript code, including several search engines, social networks and on-line shops. The *Surfing* data set comprises 20,283 URLs of web pages visited during usual web surfing at our institute. The data has been recorded over a period of 10 days and contains individual sessions of five users. Both data sets have been sanitized by scanning the web pages for drive-by downloads using common attack strings and the GoogleSafeBrowsing service. While very few unknown attacks might still be present in the data, we rely on the ability of the SVM learning algorithm to compensate this inconsistency effectively.

(a) Benign data sets

Data set	# URLs
Alexa-200k	200,000
Surfing	20,283

(b) Attack data sets

Data set	# attacks
Spam Trap	256
SQL Injection	22
Malware Forum	201
Wepawet-new	46
Obfuscated	84

Table 1: Description of benign and attack data sets. The attack data sets have been taken from [4].

[3]Alexa Top Sites, `http://www.alexa.com/topsites`

The attack data sets are listed in Table 1(b) and have been mainly taken from Cova et al. [4]. In total, the attack data sets comprise 609 samples containing several types of drive-by-download attacks collected over a period of two years. The attacks are organized according to their origin: the *Spam Trap* set comprises attacks extracted from URLs in spam messages, the *SQL Injection* set contains drive-by downloads injected into benign web sites, the *Malware Forum* set covers attacks published in Internet forums, and the *Wepawet-new* set contains malicious JavaScript code submitted to the Wepawet service[4]. A detailed description of these classes is provided in [4]. Moreover, we provide the *Obfuscated* set which contains 28 attacks from the other sets additionally obfuscated using a popular JavaScript packer[5].

3.2 Detection Performance

In our first experiment, we study the detection performance of Cujo in terms of true-positive rate (ratio of detected attacks) and false-positive rate (ratio of misclassified benign web pages). As the learning-based detection implemented in Cujo requires a set of known attacks and benign data for training detection models, we conduct the following experimental procedure: We randomly split all data sets into a *known partition* (75%) and an *unknown partition* (25%). The detection models and respective parameters, such as the best length of q-grams, are determined on the known partition, whereas the unknown partition is only used for measuring the final detection performance. We repeat this procedure 10 times and average results. The partitioning ensures that reported results only refer to attacks unknown during the learning phase of Cujo.

For comparing the performance of Cujo with state-of-the-art methods, we also consider static detection methods, namely the anti-virus scanner ClamAv[6] and the web proxy of the security suite AntiVir[7]. As ClamAv does not provide any proxy capabilities, we manually feed the downloaded web pages and respective JavaScript code to the scanner. Moreover, we add results presented by Cova et al. [4] for the offline analysis system Jsand to our evaluation.

3.2.1 True-positive Rates

Table 2 and 3 show the detection performance in terms of true-positive rates for Cujo and the other methods. The static and dynamic code analysis of Cujo alone attain a true-positive rate of 90.2% and 86.0%, respectively. The combination of both, however, allows to identify 94.4% of the attacks, demonstrating the advantage of two complementary views on JavaScript code.

A better performance is only achieved by Jsand which is able to almost perfectly detect all attacks. However, Jsand generally operates offline and spends considerably more time for analysis of JavaScript code. The anti-virus tools, ClamAv and AntiVir, achieve lower detection rates of 35% and 70%, respectively, although both have been equipped with up-to-date signatures. These results clearly confirm the need for alternative detection techniques, as provided by Cujo and Jsand, for successfully defending against the threat of drive-by-download attacks.

[4]Wepawet Service, `http://wepawet.cs.ucsb.edu`
[5]JavaScript Compressor, `http://dean.edwards.name/packer`
[6]Clam AntiVirus, `http://www.clamav.net/`
[7]Avira AntiVir Premium, `http://www.avira.com/`

Attack data sets	Cujo		
	static	dynamic	combined
Spam Trap	96.9%	98.1%	99.4%
SQL Injection	93.8%	88.3%	98.3%
Malware Forum	78.7%	71.2%	85.5%
Wepawet-new	86.3%	84.1%	94.8%
Obfuscated	100.0%	87.3%	100.0%
Average	90.2%	86.0%	94.4%

Table 2: True-positive rates of Cujo on the attack data sets. Results have been averaged over 10 runs.

Attack data sets	ClamAv	AntiVir	Jsand [4]
Spam Trap	41.0%	58.2%	99.7%
SQL Injection	18.2%	95.5%	100.0%
Malware Forum	45.3%	83.1%	99.6%
Wepawet-new	19.6%	93.5%	—
Wepawet-old	—	—	100.0%
Obfuscated	4.8%	54.8%	—
Average	35.0%	70.0%	99.8%

Table 3: True-positive rates of ClamAV, AntiVir and Jsand on the attack data sets. The Wepawet-new data set is a recent version of Wepawet-old.

3.2.2 *False-positive Rates*

Table 4 and 5 show the false-positive rates on the benign data sets for all detection methods. Except for AntiVir all methods attain reasonably low false-positive rates. The combined analysis of Cujo yields a false-positive rate of 0.002%, corresponding to 2 false alarms in 100,000 visited web sites, on the Alexa-200k data set. Moreover, Cujo does not trigger any false alarms on the Surfing data set.

The high false-positive rate of AntiVir with 0.087% is due to overly generic detection rules. The majority of false alarms shows the label `HTML/Redirector.X`, indicating a potential redirect, where the remaining alerts have generic labels, such as `HTML/Crypted.Gen` and `HTML/Downloader.Gen`. We carefully verified each of these alerts using a client-based honeypot [21], but could not determine any malicious activity on the indicated web pages.

For the false alarms raised by Cujo we identify two main causes: 0.001% of the web pages in the Alexa-200k data set contain fully encrypted JavaScript code with no plain-text operations except for `unescape` and `eval`. This drastic form of obfuscation induces the false alarms of the static analysis. The 0.001% false positives of the dynamic analysis result from web pages redirecting error messages of JavaScript to customized functions. Such redirection is frequently used in drive-by downloads to hide errors during exploitation of vulnerabilities, though it is applied in a benign context in these 0.001% cases.

Overall, this experiment demonstrates the excellent detection performance of Cujo which identifies the vast majority of drive-by downloads with very few false alarms—although all attacks have been unknown to the system. Cujo thereby significantly outperforms current anti-virus tools and is almost on par with the offline analysis system Jsand.

3.3 Explanations

After studying the detection accuracy of Cujo, we explore its ability to equip alerts with explanations, which provides a valuable instrument for analysis of detected attacks. In particular, we present explanations for the detection techniques detailed in Section 2.3 using q-grams of static and dynamic analysis reports, where we select the best q for each analysis type from the previous experiment.

Benign data sets	Cujo		
	static	dynamic	combined
Alexa-200k	0.001%	0.001%	0.002%
Surfing	0.000%	0.000%	0.000%

Table 4: False-positive rates of Cujo on the benign data sets. Results have been averaged over 10 runs.

Benign data sets	ClamAv	AntiVir	Jsand [4]
Alexa-200k	0.000%	0.087%	—
Surfing	0.000%	0.000%	—
Cova et al.	—	—	0.013%

Table 5: False-positive rates of ClamAV, AntiVir and Jsand on the benign data sets.

As the first examples, we consider the q-grams (4-grams) reported by Cujo for the static analysis of two detected drive-by downloads. Figure 6(a) shows the top five q-grams contributing to the detection of a heap-spraying attack. Some patterns indicative for this attack type are clearly visible: the first q-grams match a loop involving strings, while the last q-grams reflect an empty try-catch block. Both patterns are regularly seen in heap spraying, where the loop performs the actual spraying and the try-catch block is used for inhibiting exceptions during memory corruption.

Figure 6(b) shows the q-grams reported for the static detection of an obfuscated drive-by download. At the first glance, the top q-grams indicate only little malicious activity. However, they reveal the presence of a XOR-based decryption routine. Patterns of a loop, the XOR operator and a call to the `EVAL` function here jointly contribute to the detection of the obfuscation.

Contribution $\phi_s(x) \cdot w_s$	Features $s \in S$ (4-grams)
0.044	`+ STR.01 , STR.01`
0.043	`WHILE ( ID .`
0.042	`= ID + ID`
0.039	`{ TRY { VAR`
0.039	`) { } }`

(a) Top q-grams of a heap-spraying attack

Contribution $\phi_s(x) \cdot w_s$	Features $s \in S$ (4-grams)
0.124	`= ID + ID`
0.121	`; EVAL ( ID`
0.112	`( ID ) ^`
0.104	`) ; } ;`
0.096	`STR.01 ; FOR (`

(b) Top q-grams of an obfuscated attack

Figure 6: Examples for the explanation of static detection. The five q-grams with highest contribution to the detection are presented.

As examples for the dynamic analysis, Figure 7(a) shows the top q-grams (3-grams) contributing to the dynamic detection of a heap-spraying attack. Again the attack type is clearly manifested: the first q-gram corresponds to the abstract operation `HEAP SPRAYING DETECTED` which is triggered

Contribution $\phi_s(x) \cdot w_s$	Features $s \in S$ (3-grams)
0.190	`HEAP SPRAYING DETECTED`
0.121	`CALL unescape SET`
0.053	`SET global.shellcode TO`
0.053	`unescape SET global.shellcode`
0.036	`TO "%90%90%90%90%90%90%90...`

(a) Top q-grams of a heap-spraying attack

Contribution $\phi_s(x) \cdot w_s$	Features $s \in S$ (3-grams)
0.036	`CALL unescape CALL`
0.030	`CALL fromCharCode CALL`
0.025	`CALL eval CONVERT`
0.024	`parseInt CALL fromCharCode`
0.024	`CALL createElement ("object")`

(b) Top q-grams of an obfuscated attack

Figure 7: Examples for the explanation of dynamic detection. The five q-grams with highest contribution to the detection are presented.

by our sandbox and indicates unusual memory activity. The remaining q-grams reflect typical patterns of a shellcode construction, including the unescaping of an encoded string and a so-called NOP sled.

A further example for dynamic detection is presented in Figure 7(b), which shows the top five q-grams of an obfuscated attack. Several calls of functions typical for obfuscation and corresponding substitution ciphers are visible, including `eval` and `unescape` as well as the conversion functions `parseInt` and `fromCharCode` used during decryption of the attack. The last q-gram reflects the instantiation of an object likely related to a vulnerability in a browser extension, though the actual details of this exploitation are not covered by the first five q-grams.

It is important to note that these explanations are specific to the detection of individual attacks and must not be interpreted as stand-alone detection rules. While we have only shown the top q-grams for explanation, the underlying detection models involve several million different q-grams and thus realize a far more complex decision function.

3.4 Run-time Performance

Given the accurate detection of drive-by downloads, it remains to show that Cujo provides sufficient run-time performance for practical application. Hence, we first examine the individual run-time of each system component individually and then study the overall processing time in a real application setting with multiple users. All run-time experiments are conducted on a system with an Intel Core 2 Duo 3 GHz processor and 4 Gigabytes of memory.

3.4.1 Run-time of Components

For the first analysis, we split the total run-time of Cujo into the contributions of individual components as depicted in Figure 1. For this, we add extra timing information to the JavaScript analysis, the feature extraction and learning-based detection. We then measure the exact contributions to the total run-time on a sample of 10,000 URLs from the Alexa-200k data set.

Figure 8 shows the median run-time per URL in milliseconds, including loading of a web page, pre-loading of

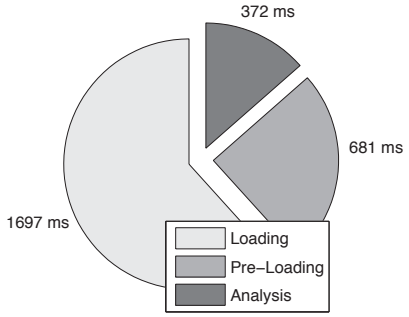


Figure 8: Median run-time of Cujo per URL on 10,000 URLs from the Alexa-200k data set.

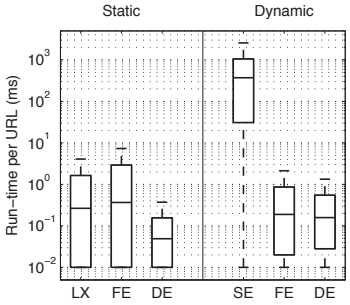


Figure 9: Statistical breakdown of run-time for JavaScript lexing (LX), sandbox emulation (SE), feature extraction (FE) and detection (DE).

external JavaScript code and the actual analysis of Cujo. Surprisingly, most of the time is spent for loading and pre-loading of content, whereas only 14% is devoted to the analysis part of Cujo. As we will see in the following section, we can greatly benefit from this imbalance by employing regular caching techniques.

A detailed statistical breakdown of the analysis run-time is presented in Figure 9, where the distributions of run-time per URL are plotted for the static and dynamic analysis separately. Each distribution is displayed as a boxplot, in which the box itself represents 50% of the data and the lower and upper markers the minimum and maximum run-time per URL. Additionally, the median is given as a middle line in each box. Except for the sandbox emulation, all components induce a very small run-time overhead ranging between 0.01 and 10 ms per URL. The sandbox analysis requires a median run-time of 370 ms per URL which is costly but still significantly faster then related sandbox approaches.

3.4.2 Operating Run-time

In the last experiment, we evaluate the run-time of Cujo in a real application setting. In particular, we deploy Cujo as a web proxy and measure the time required per delivery of a web page. To obtain reproducible measurements, we use the Surfing data set as basis for this experiment, as it contains multiple surfing sessions of five individual users. For comparison, we also employ a regular web proxy, which just forwards data to the users. As most of the total run-time is spent for loading and pre-loading of resources, we enable all caching capabilities in Cujo and the regular proxy.

Results for this experiment are shown in Figure 10, where the distribution of run-time per URL is presented as a den-

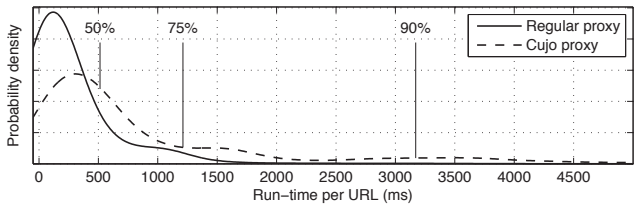


Figure 10: Operating run-time of Cujo and a regular web proxy on the Surfing data set.

sity plot. As expected the regular proxy ranges in the front part of the plot with a median processing speed of roughly 150 ms per request. The run-time of Cujo is slightly shifted to the right in comparison with the regular proxy. However, the median run-time lies around 500 ms per web page, thus inducing only a minimal delay at the web client. For example, the median run-time for visiting web pages from the domains `google.com` and `yahoo.com` using Cujo is 460 ms and 266 ms, respectively.

In contrast to the regular proxy, the run-time distribution of Cujo shows an elongated tail, where few web pages require more than 3,000 ms for processing due to excessive analysis of JavaScript code. For instance, visiting pages from `facebook.com` induces a median run-time of 1,560 ms. Still, this experiment demonstrates that Cujo strongly benefits from caching capabilities, such that only a minor delay can be perceived at the web client.

4. RELATED WORK

Since the first discovery of drive-by downloads, analysis and detection of this threat has been a vital topic in security research. One of the first studies on these attacks and respective defenses has been conducted by Provos et al. [15, 16]. The authors inspect web pages by monitoring a web browser for anomalous activity in a virtual machine. This setup allows for detecting a broad range of attacks. However, the analysis requires prohibitive run-time for on-line application, as the virtual machine needs to be restored and run for each web page individually.

A similar approach for identification of drive-by downloads is realized by client-based honeypots, such as Capture-HPC [21] and PhoneyC [14]. While Capture-HPC also relies on monitoring state changes in a virtual machine, PhoneyC emulates known vulnerabilities to capture attacks in a lightweight manner. Although effective in identifying web pages with malicious content, client-based honeypots are designed for offline analysis and thus suffer from considerable run-time overhead.

In contrast to these generic techniques, other approaches focus on identifying particular attacks types, namely heap-spraying attacks. For example, the system Nozzle proposed by Ratanaworabhan et al. [17] intercepts the memory management of a browser for detecting valid x86 code in heap objects. Similarly, Egele et al. [8] instrument SpiderMonkey for scanning JavaScript strings for the presence of executable x86 code. Both systems provide an accurate and efficient detection of heap-spraying attacks, yet they fail to identify other common types of drive-by-download attacks, for example, using insecure interfaces of browser extensions for infection.

Closest to our work is the analysis system Jsand developed by Cova et al. [4] as part of the Wepawet service. Jsand analyses JavaScript using the framework HtmlUnit and the interpreter Rhino which enable the emulation of an entire browser environment and monitoring of sophisticated interaction with the DOM tree. The recorded behavior is analysed using 10 features specific to drive-by-download attacks for anomalous activity. Due to its public web interface, Jsand is frequently used by security researchers to study novel attacks and has proven to be a valuable analysis instrument. However, its broad analysis of JavaScript code is costly and induces a prohibitive average run-time of about 25 seconds per web page [cf. 4].

Finally, the system Noxes devised by Kirda et al. [12] implements a web proxy for preventing cross-site scripting attacks. Although not directly related to this work, Noxes is a good example of how a proxy system can transparently protect users from malicious web content. Obviously, this approach targets only cross-site scripting attacks and does not protect from other threats, such as drive-by downloads.

5. CONCLUSIONS

In this paper, we have presented Cujo, a system for effective and efficient prevention of drive-by downloads. As an extension to a web proxy, Cujo transparently inspects web pages using static and dynamic detection models and allows for blocking malicious code prior to delivery to the client. In an empirical evaluation with 200,000 web pages and 600 drive-by-download attacks, a prototype of this system significantly outperforms current anti-virus products and enables detecting 94% of the drive-by downloads with few false alarms and a median run-time of 500 ms per web page—a delay hardly perceived at the web client.

While the proposed system does not generally eliminate the threat of drive-by downloads, it considerably raises the bar for adversaries to infect client systems. To further harden this defense, we currently investigate combining Cujo with offline analysis and honeypot systems. For example, malicious code detected using honeypots might be directly added to the training data of Cujo for keeping detection models up-to-date. Similarly, offline analysis might be applied for inspecting and explaining detected attacks in practice.

Acknowledgements

The authors would like to thank Marco Cova for providing the attack data sets as well as Martin Johns and Thorsten Holz for fruitful discussions on malicious JavaScript code and its detection.

References

[1] Standard ECMA-262: ECMAScript Language Specification (JavaScript). 3rd Edition, ECMA International, 1999.

[2] Symantec Global Internet Security Threat Report: Trends for 2009. Vol. XIV, Symantec, Inc., 2010.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1985.

[4] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious

JavaScript code. In *Proc. of the International World Wide Web Conference (WWW)*, 2010.

[5] M. Daniel, J. Honoroff, and C. Miller. Engineering heap overflow exploits with JavaScript. In *Proc. of USENIX Workshop on Offensive Technologies (WOOT)*, 2008.

[6] A. Dewald, T. Holz, and F. Freiling. ADSandbox: Sandboxing JavaScript to fight malicious websites. In *Proc. of ACM Symposium on Applied Computing (SAC)*, 2010.

[7] M. Egele, E. Kirda, and C. Kruegel. Mitigating drive-by download attacks: Challenges and open problems. In *Proc. of Open Research Problems in Network Security Workshop (iNetSec)*, 2009.

[8] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2009.

[9] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

[10] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for Unix processes. In *Proc. of IEEE Symposium on Security and Privacy*, pages 120–128, Oakland, CA, USA, 1996.

[11] M. Johns. On JavaScript malware and related threats – Web page based attacks revisited. *Journal in Computer Virology*, 4(3):161–178, 2008.

[12] E. Kirda, C. Kruegel, G. Vigna, , and N. Jovanovic. Noxes: A client-side solution for mitigating cross site scripting attacks. In *Proc. of ACM Symposium on Applied Computing (SAC)*, 2006.

[13] K.-R. Müller, S. Mika, G. Rätsch, K. Tsuda, and B. Schölkopf. An introduction to kernel-based learning algorithms. *IEEE Neural Networks*, 12(2):181–201, May 2001.

[14] J. Nazario. A virtual client honeypot. In *Proc. of USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.

[15] N. Provos, P. Mavrommatis, M. Rajab, and F. Monrose. All your `iframes` point to us. In *Proc. of USENIX Security Symposium*, 2008.

[16] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser: Analysis of web-based malware. In *Proc. of USENIX Workshop on Hot Topics in Understanding Botnets (HotBots)*, 2007.

[17] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. Technical Report MSR-TR-2008-176, Microsoft Research, 2008.

[18] K. Rieck and P. Laskov. Detecting unknown network attacks using language models. In *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 74–90, July 2006.

[19] K. Rieck and P. Laskov. Linear-time computation of similarity measures for sequential data. *Journal of Machine Learning Research*, 9(Jan):23–48, 2008.

[20] B. Schölkopf and A. Smola. *Learning with Kernels*. MIT Press, Cambridge, MA, 2002.

[21] C. Seifert and R. Steenson. Capture – honeypot client (Capture-HPC). Victoria University of Wellington, NZ, https://projects.honeynet.org/capture-hpc, 2006.

[22] K. Wang, J. Parekh, and S. Stolfo. Anagram: A content anomaly detector resistant to mimicry attack. In *Recent Advances in Intrusion Detection (RAID)*, pages 226–248, 2006.

Fast and Practical Instruction-Set Randomization for Commodity Systems

Georgios Portokalidis and Angelos D. Keromytis
Network Security Lab
Department of Computer Science
Columbia University, New York, NY, USA
{porto, angelos}@cs.columbia.edu

ABSTRACT

Instruction-set randomization (ISR) is a technique based on randomizing the "language" understood by a system to protect it from code-injection attacks. Such attacks were used by many computer worms in the past, but still pose a threat as it was confirmed by the recent Conficker worm outbreak, and the latest exploits targeting some of Adobe's most popular products. This paper presents a fast and practical implementation of ISR that can be applied on currently deployed software. Our solution builds on a binary instrumentation tool to provide an ISR-enabled execution environment entirely in software. Applications are randomized using a simple XOR function and a 16-bit key that is randomly generated every time an application is launched. Shared libraries can be also randomized using separate keys, and their randomized versions can be used by all applications running under ISR. Moreover, we introduce a key management system to keep track of the keys used in the system. *To the best of our knowledge we are the first to apply ISR on truly shared libraries.*

Finally, we evaluate our implementation using real applications including the Apache web server, and the MySQL database server. For the first, we show that our implementation has negligible overhead (less than 1%) for static HTML loads, while the overhead when running MySQL can be as low as 75%. We see that our system can be used with little cost with I/O intensive network applications, while it can also be a good candidate for deployment with CPU intensive applications, in scenarios where security outweighs performance.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Invasive software

General Terms

Security, Reliability, Performance

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

Keywords

Code-injection, randomization, security, performance

1. INTRODUCTION

Instruction-set randomization [25, 4] is a technique based on randomizing a system's language (*i.e.,* its instruction set) to prevent code-injection attacks. Such attacks occur when the attacker is able to execute arbitrary code remotely, or locally but as a different user (*e.g.,* a user with elevated privileges like the administrator). They usually follow the exploitation of buffer overflows [35, 3, 27] and other memory corruption vulnerabilities, like dangling pointers [20, 34] and format string attacks [39], that allow attackers to redirect execution to the injected code.

In the past, code-injection attacks (CI) accounted for almost half of the advisories released by CERT [43], and were used by many computer worms to infect new hosts [41, 11, 12, 29, 44]. More recently, they have been overshadowed by cross-site scripting and SQL-injection attacks, but the recent Conficker worm outbreak [36], and the multiple vulnerabilities discovered in Adobe's popular software [1, 42] came as a reminder that CI attacks still pose a significant threat to a large number of systems.

ISR is a general approach that defeats *all types* of remote code-injection regardless of the way it was injected into a process. It operates by randomizing the instructions that the underlying system "understands", so that "foreign" code such as the code injected during an attack will fail to execute. It was initially proposed as a modification to the processor to ensure low performance overheads, but unfortunately this proposal has had little allure with hardware vendors. Instead, software implementations of ISR on x86 emulators have been created, mainly to demonstrate the effectiveness of the approach, as they incur large runtime overheads [25, 4]. Software only implementations of ISR using dynamic binary translation have been also proposed [24], but have seen little use in practice as they cannot be directly applied to commodity systems. For instance, they do not support shared libraries or dynamically loaded libraries (*i.e.,* they require that the application is statically linked), and increase the code size of encoded applications.

This paper describes a fast and practical software implementation of ISR for commodity systems. Our implementation is based on Intel's dynamic instrumentation tool called PIN [26], which provides the runtime environment. Application code is randomized using the XOR function and a 16-bit key, which is randomly generated every time the ap-

plication is launched to make it resistant against key guessing attacks [40].

Multiple keys can be used to randomize different parts of the application. For instance, every shared library used by the system can be randomized using a different key, creating a randomized copy of each library. While additional disk space will be required to store the randomized versions, during runtime all binaries running under ISR will be using the same randomized copy. Also, original (native) code can be combined with randomized code. The keys used to encode the various libraries are managed using SQLite [32], a self-contained and serverless database engine. Libraries can be randomized once and reused by multiple applications, while frequently re-randomizing them also protects them against key guessing attempts. Finally, we assume (as does past work) that the attacker does not have access to the randomized code (*i.e.,* it is a remote attacker), so a known ciphertext attack against the key is not possible.

The *main contributions* of this paper can be summarized in the following:

- We implemented instruction-set randomization for commodity systems using Intel's PIN framework (our implementation of ISR is freely available from `https://sourceforge.net/projects/isrupin/`)
- Our implementation operates on currently deployed binaries, as it does not require recompilation, or changes to the underlying system (*i.e.,* the operating system and hardware)
- Our system supports dynamically linked executables, as well as dynamically loaded libraries. We also introduce a key management scheme for storing and keeping track of the keys used to randomize shared libraries and applications. To the best of our knowledge we are the first to apply ISR on shared libraries
- Executables are re-randomized every time they are launched, and shared libraries are re-randomized at custom intervals to protect the key from guessing attacks such as [40]

The overhead of our implementation can be as low as 10% compared with native execution. We are able to run popular servers such as the Apache web server, and the MySQL database server, and show that running Apache using ISR has negligible effect on throughput for static HTML loads, while the overhead for running MySQL is 75%. We also evaluate the cost of completely isolating the framework's data from the application. This memory protection (MP) requires more invasive instrumentation of the target application, and it has not been investigated by previous work on software-based ISR, since it incurs significant overhead. We show that adding MP over ISR does not reduce Apache's throughput, while it imposes an extra 57% overhead when running MySQL.

The rest of this paper is organized as follows: Section 2 offers a brief description of ISR. Our implementation is discussed in Section 3. We evaluate the performance of our framework in Section 4. Related work is examined in Section 5. Finally, conclusions are in Section 6.

2. INSTRUCTION-SET RANDOMIZATION

Instruction-set randomization as a mean to thwart code-injection attacks has been presented in detail in previous work [25, 4]. In this section we will only briefly describe the technique, mainly focusing on its application on binaries.

ISR is based on the observation that code-injection attacks need to position executable code within the address space of the exploited application and then redirect control to it. The injected code needs to be compatible with the execution environment for these attacks to succeed. In other words, the attacker needs to be able to "talk" to the target system in its own "language". For binary programs, this means that the code needs to be compatible with the processor and software running at the target. For instance, injecting x86 code into a process running on an ARM system will most probably cause it to crash, either because of an illegal instruction being executed, or due to an illegal memory access. We should note that in this example it is possible to compose (somewhat limited) machine code able to run without errors on both ARM and x86.

ISR builds on this observation to block attackers from executing code injected in vulnerable processes. An execution environment employing a randomly generated instruction set is used to run processes, causing injected code to fail. While exploitation attempts will still cause a DoS by crashing the targeted application, attackers are not able to perform any useful action such as installing malware or rootkits. The strength of the technique lies in the difficulty of guessing the instruction set used by a process. Of course, if an attacker has access to the randomized binary, he can launch an attack against the applied transformation to attempt to learn the new instruction set, something that requires local access to the target host. This work (and ISR in general) is primarily focused on protecting against remote attacks on network services (*e.g.,* http, dns, ssh, *etc.*), where the attacker does not have access to the target system or the randomized binaries. Consequently, attackers cannot launch attacks against the key that require access to the ciphertext.

However, remote attackers can still attempt to guess the key used to randomize the instruction set [40]. Such guessing attacks will cause the application to crash and restart for each failed attempt to correctly guess the key. We can mitigate such attacks by either using a more complicated encoding algorithm (*e.g.,* bit transposition, AES, *etc.*) and a larger key to increase the complexity of the attack, or by frequently re-encoding the binary using a new key every time it is executed as we discuss below. The reader can refer to our earlier work on ISR [25] for additional discussion on randomization using larger keys.

2.1 ISR Operation

CPU instructions for common architectures, like x86 and ARM, consist of two parts: the *opcode* and *operands*. The opcode defines the action to be performed, while the operands are the arguments. For example, in the the x86 architecture a software interrupt instruction (`INT`) comprises of the opcode `0xCD`, followed by a one-byte operand that specifies the type of interrupt. We can create new instruction sets by randomly creating new mappings between opcodes and actions. We can further randomize the instruction set by also including the operands in the transformation.

For ISR to be effective and efficient, the number of possible instruction sets must be large, and the mapping between the

new opcodes and instructions should be efficient (*i.e.,* not completely arbitrary). We can achieve both these properties by employing cryptographic algorithms and a randomly generated secret key. As an example, consider a generic RISC processor with fixed-length 32-bit instructions. We can effectively generate random instruction sets by encoding instructions with XOR and a secret 32-bit key. In this example, an attacker would have to try 2^{32} combinations in the worst case to guess the key. Architectures with larger instructions (*i.e.,* 64 bits) can use longer keys to be even more resistant to brute-force attacks. On the other hand, simply increasing the length of the key used with XOR will not improve security, since the key can be attacked in a piece-meal fashion (*i.e.,* by guessing the first 32 bits of the key that correspond to a single instruction). *The situation is even more complicated on architectures with variable sized instructions like the x86.* Many instructions in the x86 architecture are 1 or 2 bytes long. This effectively splits the key in four or two sub-keys of 8 and 16 bits respectively. Thus, it is possible that an attacker attempts to guess each of the sub-keys independently, as shown by Sovarel *et al.* [40].

The deficiencies of XOR randomization on architectures like the x86 can be overcome by using other ciphers for randomizing instructions. For instance, bit transposition of larger blocks (*e.g.,* 160 bits) would greatly increase the work factor for an attacker, and cannot be attacked in a piece-meal fashion. Hu et al [24] propose the use of AES encryption on blocks of 128 bits to ensure that an attacker cannot break the randomization. In both cases larger blocks of data need to be accessible at runtime, and more processing is required to decode the instructions. We have taken a different approach to protect the keys. First, we employ multiple keys for the encoding of an application (*i.e.,* a different key for each shared library). Second, we randomize an application every time it is launched with a new random key, and third we frequently re-randomize shared libraries.

Finally, we note that the security of the approach depends on the fact that injected code will raise an exception (*e.g.,* by accessing an illegal address or using an invalid opcode), after it has been de-randomized by the execution environment. While this will generally be true, there are a few permutations of injected code that will result in working code that performs the attacker's task. This number is statistically insignificant [5].

2.2 ISR Runtime

A randomized process requires the appropriate execution environment to de-randomize its instructions before they are executed. Previous work on ISR has demonstrated that it is possible to implement such an environment both in hardware and software. In both cases, the environment needs access to the key used during the randomization. The key can be stored within the executable, or in a database. Storing it within the application is compact and removes the need for external storage (*i.e.,* a DB), but could expose the key if the application leaks information.

Additionally, programs frequently make use of libraries, which may or may not be randomized. ISR needs to be able to detect when execution switches from a randomized piece of code to a plain one, and vice-versa. Detecting such context switches can be complex (specially in hardware), and in fact previous work has only handled statically linked executables. We will show in Section 3 that *our implementation is able to handle dynamically linked applications by supporting multiple instruction sets per process (i.e., instructions randomized with different keys).*

3. IMPLEMENTATION

We implemented ISR in software on 32-bit Linux for dynamically and statically linked ELF executables and libraries. This section describes the components of our implementation. It should be noted that while the current implementation works on Linux, it can be easily ported to other platforms also supported by the runtime.

3.1 Randomization of Binaries

ELF (the executable and linking format) is a very common and standard file format used for executables and shared libraries in many Unix type systems like Linux, BSD, Solaris, *etc.* Despite the fact that it is most commonly found on Unix systems, it is very flexible and it is not bound to any particular architecture or OS. Also, the ELF format completely separates code and data, including control data such as the procedure linkage table (PLT), making it an ideal format for applying binary randomization.

We modified the `objcopy` utility, which is part of the GNU binutils package to add support for randomizing ELF executables and libraries. `objcopy` can be used to perform certain transformations (*e.g.,* strip debugging symbols) on an object file, or simply copy it to another. Thus, it is able to parse the ELF headers of an executable or library and access its code. We modified `objcopy` to randomize a file's code using XOR and a 16-bit key. We also extended `objcopy` to randomize shared libraries in ELF format. Randomizing using XOR does not require that the target binary is aligned, so it does not increase its size or modify its layout.

While our current implementation is currently able to randomize only ELF binaries, support for other binaries can be easily added. For instance, we plan to extend `objcopy` to also randomize *Portable Executable (PE)* binaries for Windows operating systems [28].

3.2 Shared Libraries

Most executables in modern OSs are dynamically linked to one or more shared libraries. Shared libraries are preferred because they accommodate code reuse and minimize memory consumption, as their code can be concurrently mapped and used by multiple applications. As a result, mixing shared libraries with ISR has proven to be problematic in past work. Our implementation of ISR supports multiple instruction sets (*i.e.,* multiple randomization keys) for the same process, enabling us to use plain shared libraries with a randomized executable. Furthermore, it enables us to randomize each library with its own key, and share it amongst all processes running under ISR like an ordinary shared library.

We create a randomized copy of all libraries that are needed, and store them in a shadow folder (*e.g.,* "/usr/rand_lib"). For stronger security, each library is encoded using a different key, while we can also periodically re-randomize all the libraries using new keys. When an application is loaded in the runtime environment, we modify its environment so it first looks for shared libraries in a shadow folder. If a randomized version of a library is not found, it proceeds to look for a plain version in the usual system locations (*e.g.,* "/usr/lib" and "/lib" on Linux, and "c:\windows\system32"

for Windows). Of course, a process can be forced to only use randomized code if that is required. Moreover, multiple shadow folders can be used concurrently. For instance, if a process crashes (*e.g.,* a crash could be triggered by a failed exploitation attempt), we may re-encode all shared libraries to thwart key guessing attacks.

3.3 Key Management

Supporting multiple instruction sets for every process notably increases the number of keys that are active in the system at any given time. Thus, key management becomes an important aspect of the system, and specially because shared libraries can be randomized with their own key, and multiple versions of the libraries may co-exist in the system. Previous work proposed to store keys within the ELF files, which removes the need for separate storage for the keys. While this approach is robust, it leaves keys vulnerable to exposure if an application leaks data because of a bug or an error. In the past information leakage has been exploited to bypass address space layout randomization (ASLR) [19]. Additionally, storing the key within the executable might not be feasible when using binary formats other than ELF.

Instead, we store the keys for executables and libraries in a database, using the *sqlite* database system. *Sqlite* is a software library that implements a self-contained, serverless SQL database engine. The entire database is stored in a single file, and it is accessed directly by our tool (using the *sqlite* library) without the need to run additional processes. The keys are indexed using the library's full path, and the operation of retrieving a key from the DB is fast. As it is an operation that it is only performed when an application is launched or a dynamic library is loaded, its performance is not critical for the system.

3.4 PIN Execution Environment

We implemented the de-randomizing execution environment using Intel's dynamic binary instrumentation tool PIN. PIN [26] is an extremely versatile tool that operates entirely in user-space, and supports multiple architectures (x86, 64-bit x86, ARM) and operating systems (Linux, Windows, MacOS). It operates by just-in-time (JIT) compiling the target's instructions combined with any instrumentation into new code, which is placed into a code cache, and executed from there. It also offers a rich API to inspect and modify an application's original instructions.

We make use of the supplied API to implement our ISR runtime framework. First, we install a callback that intercepts the loading of all file images. This provides us with the names of all the shared libraries being used, as well as the memory ranges where they have been loaded in the address space. We use the path and name of the library to lookup its key in the database and load it. We save the library's key and memory address range in a hash table-like data structure that allows us to quickly lookup a key using a memory address. The existence of a key in the database also indicates that the library is encoded, so no special handling is required to load system libraries (*i.e.,* not encoded libraries).

The actual de-randomization is performed by installing a callback that replaces PIN's default function for fetching code from the target process. This second callback reads instructions from memory, and uses the memory address to lookup the key to use for decoding. If the instruction fetched is within the memory range of a shared library we use its key for decoding, or assume no decoding is necessary if no key is present. All instructions not associated with a library are considered to be part of the executable and are decoded using its key. To avoid performing a lookup for every instruction fetched, we cache the last used key. During our evaluation this simple single entry cache achieved high hit ratios, so we did not explore other caching mechanisms.

3.5 Memory Protection (MP)

When executing an application within PIN, they both operate on the same address space. This means that in theory an application can access and modify the data used by PIN and consequently ISR. Such illegal accesses may occur due to a program error, and could potentially be exploited by an attacker. For instance, an attacker could attempt to overwrite a function pointer or return address in PIN, so that control is diverted directly into the attacker's code in the application. Such a control transfer would circumvent ISR enabling the attacker to successfully execute his code. To defend against such attacks we need to protect PIN's memory from being written by the application.

When PIN loads and before the target application and its libraries gets loaded, we scan the address space to identify all memory pages used by PIN. We mark these memory pages by asserting a flag in an array (*page-map*), which holds one byte for every addressable page. For instance, in a 32-bit Linux system, processes can typically access 3 out of the 4 GBytes that are directly addressable. For a page size of 4 KBytes, this corresponds to 786432 pages, so we allocate 768 KBytes to store the flags for the entire address space. At runtime, when additional memory is used by PIN, we update the flags for the newly used pages in the page-map. Memory protection is actually enforced by instrumenting all memory write operations performed by the application, and checking that the page being accessed is valid according to the page-map. If the application attempts to write to a page "owned" by PIN, the instrumentation causes a page-fault that will terminate it.

Introducing memory protection further hardens the system against code-injection attacks, but incurs a substantial overhead. However, forcing an attacker to exploit a vulnerability in this fashion is already hardening the system considerably, as he would have to somehow discover one of the few memory locations which can be used to divert PIN's control flow. Alternatively, we can use address space layout randomization to decrease the probability of an attacker successfully guessing the location of PIN's control data.

3.6 ISR Exceptions

While all instructions in the application are encoded, there are cases where certain external and unencoded instructions need to be executed in the context of the process. For instance, some systems inject code within the stack of a process when a signal is delivered. These *signal trampolines* are used to set up and clean up the context of a signal handler. The instructions are a type of legitimate code-injection performed by the system, and need special handling or their execution will lead to a crash. Fortunately, signal trampolines are very small (approximately 5-7 instructions long), and the instructions used are fixed on every system (*i.e.,* the same instructions are used for all signals in the system). When a signal is delivered to a process, we scan the code

being executed to identify *trampolines*, and execute them without applying the decoding function.

Moreover, modern Linux systems frequently include a read-only *virtual shared object (VDSO)* in every running process. This object is used to export certain kernel functions to user space. For instance, it is used to perform system calls, replacing the older software interrupt mechanism (`INT 0x80`). This object needs to be treated in the same manner as plain shared libraries, allowing the execution of non-randomized code. Since this is a read-only object, we can safely do so.

3.7 Startup Procedure

When a dynamically linked application is executed, the loader looks for shared libraries in certain predefined locations (*e.g.,* "/usr/lib", "/lib", *etc.*), as well as locations specified in the environment (*i.e.,* the environment variable `LD_LIBRARY_PATH`). To enable the loading of the randomized versions of shared libraries, we need to add the shadow folder in the search path. We cannot do so by adding the folder in the system's library search path, as that would cause these libraries to be used instead of the originals for all running applications. Instead, we use `LD_LIBRARY_PATH`. Unfortunately, as PIN itself is dynamically linked we cannot set the variable directly. We employ a wrapper program that we launch using PIN. The wrapper adds the shadow folder in the library search path, and launches the target application, which then looks for libraries in the shadow folder first.

4. PERFORMANCE

Dynamic instrumentation tools usually incur significant slowdowns on target applications. While this is also true for PIN, we show that the overhead is not prohibitive. We conducted the measurements presented in this section on a DELL Precision T5500 workstation with a dual 4-core Xeon CPU and 24GB of RAM running Linux.

Figure 1 shows the mean execution time and standard deviation when running several commonly used Linux utilities. We draw the execution time for running *ls* on a directory with approximately 3400 files, and running *cp*, *cat*, and *bunzip2* with a 64MB file. We tested four execution scenarios: native execution, execution with PIN and no instrumentation (PIN's minimal overhead), our implementation of ISR without memory protection (MP), and lastly with MP enabled (ISR-MP). The figure shows that short-lived tasks suffer more, because the time needed to encode the binary and initialize PIN is relatively large when compared with the task's lifetime. In opposition, *when executing a longer-lived task, such as bunzip2, execution under ISR only takes about 10% more time to complete.*

For all four utilities, when employing memory protection to protect PIN's memory from interference, execution takes significantly longer, with bunzip2 being the worst case requiring *almost 4 times* more time to complete. That is because memory protection introduces additional instructions at runtime to check the validity of all memory write operations. Another interesting observation is that running bunzip2 under ISR is slightly faster from just using PIN. We attribute this to the various optimizations that PIN introduces when actual instrumentation is introduced.

We also evaluate our implementation using two of the most popular open-source servers: the *Apache* web server, and the *MySQL* database server. For Apache, we measure the effect that PIN and ISR have on the maximum throughput of a static web page, using Apache's own benchmarking tool *ab* over a dedicated 1 Gb/s network link. To avoid high fluctuations in performance due to Apache forking extra processes to handle the incoming requests in the beginning of the experiment, we configured it to pre-fork all worker processes (pre-forking is a standard multi-processing Apache module), and left all other options to their default setting.

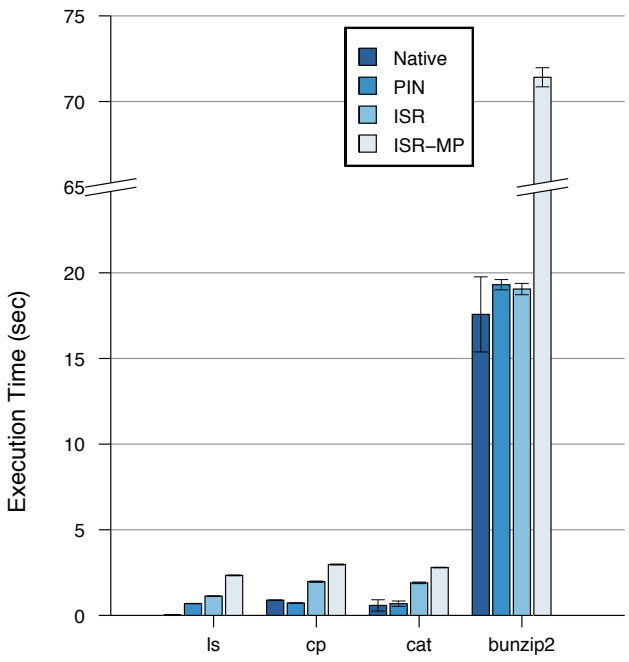


Figure 1: Execution time of basic Linux utilities. The figure draws the mean execution time and standard deviation when running four commonly used Linux utilities.

Figure 2 shows the mean throughput and standard deviation of Apache for the same four scenarios used in our first experiment. The graph shows that Apache's throughput is more limited by available network bandwidth than CPU power. Running the server over PIN has no effect on the attainable throughput, while applying ISR, even with memory protection enabled, does not affect server throughput either.

Finally, we benchmarked a MySQL database server using its own *test-insert* benchmark, which creates a table, fills it with data, and selects the data. Figure 3 shows the time needed to complete this benchmark for the same four scenarios. PIN introduces a 75% overhead compared with native execution, while our ISR implementation incurs no observable slowdown. Unlike Apache, enabling memory protection for MySQL is 57.5% slower that just using ISR (175% from native). As with Apache, the benchmark was run at a remote client over a 1 Gb/s network link to avoid interference with the server.

5. RELATED WORK

Instruction-set randomization was initially proposed as a general approach against code-injection attacks by Gaurav *et al.* [25]. They propose a low-overhead implementation of ISR in hardware, and evaluate it using the Bochs x86 emulator. They also demonstrate the applicability of the approach

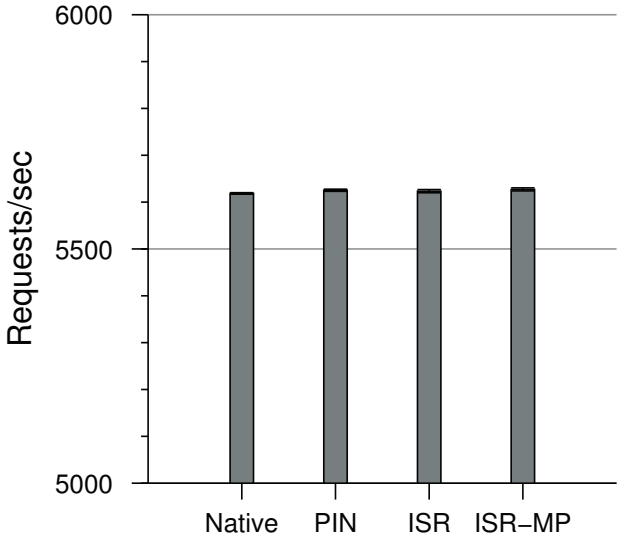


Figure 2: Apache web server throughput. The figure draws the mean reqs/sec and standard deviation as measured by Apache's benchmark utility *ab*.

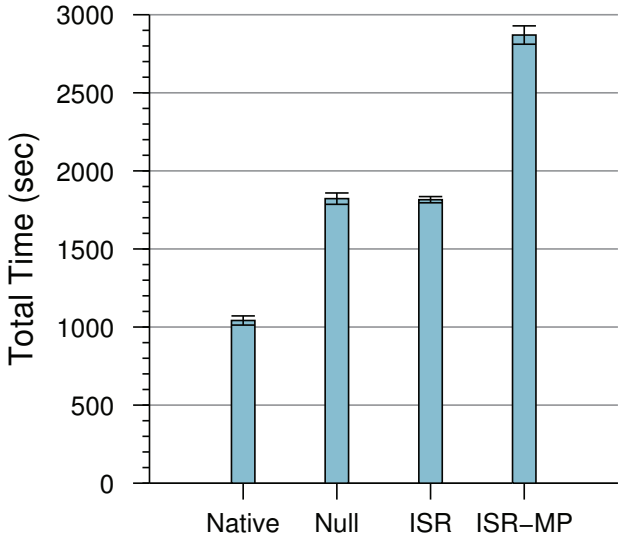


Figure 3: MySQL *test-insert* benchmark. It measures table creation, data insertion, and selection. The figure draws total execution time as reported by the benchmark utility.

on interpreted languages such as Perl, and later SQL [9]. Concurrently, Barrantes *et al.* [4] proposed a similar randomization technique for binaries (RISE), which builds on the Valgrind x86 emulator. RISE provides limited support for shared libraries by creating randomized copies of the libraries for each process. As such, the libraries are not actually shared, and consume additional memory each time they are loaded. Furthermore, Valgrind incurs a minimum performance overhead of 400% [18], which makes its use impractical.

The work closest to ours is by Hu *et al.* [24]. They also employ a virtual execution environment based on a dynamic binary translation framework named STRATA. Their implementation uses AES encryption with a 128-bit key, which requires that code segments are aligned at 128-bit blocks. Unlike our implementation, they do not support self-modifying code, and they produce randomized binaries that are significantly larger from the originals (*e.g.,* the randomized version of Apache was 77% larger than the original). Also, to the best of our knowledge previous work on ISR does not address the implications introduced by signal trampolines and VDSO, nor does it investigate the costs involved with protecting the execution environment from the hosted process (STRATA protects only a part of its data).

Address obfuscation is another approach based on randomizing the execution environment (*i.e.,* the locations of code and data) to harden software against attacks [7, 33]. It can be performed at runtime by randomizing the layout of a process (ASLR) including the stack, heap, dynamically linked libraries, static data, and the process's base address. Additionally, it can be performed at compile time to also randomize the location of program routines and variables. Shacham *et al.* [38] show that ASLR may not be very effective on 32-bit systems, as they do not allow for sufficient entropy. In contrast, Bhatkar *et al.* [8] argue that it is possible to introduce enough entropy for ASLR to be effective. Meanwhile, attackers have successfully exploited ASLR enabled systems by predicting process layout, exploiting applications to expose layout information [19], or using techniques like heap spraying [16].

Hardware extensions such as the NoExecute (NX) bit in modern processors [22, 33] can stop code-injection attacks all together without impacting performance. This is accomplished by disallowing the execution of code from memory pages that are marked with the NX bit. Unfortunately, its effectiveness is dependent on its proper use by software. For instance, many applications like browsers do not set it on all data segments. This can be due to backward compatibility constraints (*e.g.,* systems using signal trampolines), or even just bad developing practice.

PointGuard [14] uses encryption to protect pointers from buffer overflows. It encrypts pointers in memory, and decrypts them only when they are loaded to a register. It is implemented as a compiler extension, so it requires that source code is available for recompilation. Also, while it is able to deter buffer overflow attacks, it can be defeated by format string attacks that frequently employ code-injection later on. Other solutions implemented as compiler extensions include Stackguard [15] and ProPolice [21]. They operate by introducing special secret values in the stack to identify and prevent stack overflow attacks, but can be subverted [10]. Write integrity testing [2] uses static analysis and "guard" values between variables to prevent memory corruption errors, but static analysis alone cannot correctly classify all program writes. CCured [30] is a source code transformation system that adds type safety to C programs, but it incurs a significant performance overhead and is unable to statically handle some datatypes. Generally, solutions that require recompilation of software are less practical, as source code or parts of it (*e.g.,* third-party libraries) are not always available.

Dynamic binary instrumentation is used by many other solutions to retrofit unmodified binaries with defenses against

remote attacks. For instance, dynamic taint analysis (DTA) is used by many projects [31, 17, 13, 23], and is a able to detect control hijacking and code-injection attacks, but incurs large slowdowns (*e.g.,* frequently 20x or more). Due to their large overhead, dynamic solutions are mostly used for the analysis of attacks and malware [6], and in honeypots [37].

6. CONCLUSIONS

We described a fast and practical implementation of ISR based on Intel's dynamic instrumentation tool PIN. Our implementation works on commodity systems, and does not require the recompilation or relinking of target applications. Binaries are randomized at execution time, while shared libraries can be encoded beforehand and shared between the processes executing using ISR. Moreover, we introduce a simple management scheme to keep track of the randomized shared libraries and their associated keys.

Our solution operates with relatively small overhead that makes it an attractive countermeasure to retrofit security sensitive applications with. Applying it on the Apache web server has negligible effect on throughput for static HTML loads, while MySQL performs approximately 75% slower. Furthermore, we show that the overhead is largely attributed to PIN, and can be easily mitigated when applied on long-running I/O driven applications such as network services.

Acknowledgements

This work was supported by the United States Air Force Research Laboratory (AFRL) through Contract FA8650-10-C-7024 and by the National Science Foundation (NSF) through Grant CNS-09-14845. Opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US Government, the Air Force, or the NSF.

7. REFERENCES

[1] Adobe. Security advisory for flash player, adobe reader and acrobat. http://www.adobe.com/support/security/advisories/apsa10-01.html, June 2010.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 263–277, May 2008.

[3] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.

[4] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 281–289, October 2003.

[5] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović. Randomized instruction set emulation. *ACM Trans. Inf. Syst. Secur.*, 8(1):3–40, 2005.

[6] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A tool for analyzing malware. In *Proceedings of the* 15^{th} *European Institute for Computer Antivirus Research (EICAR) Annual Conference*, April 2006.

[7] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceedings of the* 12^{th} *USENIX Security Symposium*, pages 105–120, August 2003.

[8] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the* 14^{th} *USENIX Security Symposium*, pages 255–270, August 2005.

[9] S. W. Boyd, G. S. Kc, M. E. Locasto, A. D. Keromytis, and V. Prevelakis. On the general applicability of instruction-set randomization. *IEEE Transactions on Dependable and Secure Computing*, 99, 2008.

[10] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack*, 5(56), May 2000.

[11] CERT advisory CA-2001-19: "Code Red" worm exploiting buffer overflow in IIS indexing service DLL. http://www.cert.org/advisories/CA-2001-19.html, July 2001.

[12] Cert Advisory CA-2003-04: MS-SQL Server Worm. http://www.cert.org/advisories/CA-2003-04.html, January 2003.

[13] M. Costa, J. Crowcroft, M. Castro, and A. Rowstron. Vigilante: End-to-end containment of internet worms. In *Proceedings of the ACM Symposium on Systems and Operating Systems Principles (SOSP)*, October 2005.

[14] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the* 12^{th} *USENIX Security Symposium*, pages 91–104, August 2003.

[15] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the* 7^{th} *USENIX Security Symposium*, January 1998.

[16] DarkReading. Heap spraying: Attackers' latest weapon of choice. http://www.darkreading.com/security/vulnerabilities/showArticle.jhtml?articleID=221901428, November 2009.

[17] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[18] V. Developers. Valgrind user manual – callgrind. http://valgrind.org/docs/manual/cl-manual.html.

[19] T. Durden. Bypassing PaX ASLR protection. *Phrack*, 0x0b(0x3b), July 2002.

[20] C. W. Enumeration. CWE-416: use after free. http://cwe.mitre.org/data/definitions/416.html, April 2010.

[21] J. Etoh. GCC extension for protecting applications from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp/, June 2000.

[22] E. Hardware. CPU-based security: The NX bit. http://hardware.earthweb.com/chips/article.php/3358421, May 2004.

[23] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *Proceedings of the* 1^{st} *ACM EuroSys Conference*, pages 29–41, April 2006.

[24] W. Hu, J. Hiser, D. Williams, A. Filipi, J. W. Davidson, D. Evans, J. C. Knight, A. Nguyen-Tuong, and J. Rowanhill. Secure and practical defense against code-injection attacks using software dynamic

translation. In *Proceedings of the* 2^{nd} *International Conference on Virtual Execution Environments (VEE)*, pages 2–12, June 2006.

[25] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the* 10^{th} *ACM Conference on Computer and Communications Security (CCS)*, October 2003.

[26] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 190–200, June 2005.

[27] M. Conover and w00w00 Security Team. w00w00 on heap overflows. http://www.w00w00.org/files/articles/heaptut.txt.

[28] Microsoft. Microsoft portable executable and common object file format specification. http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx.

[29] D. Moore, C. Shanning, and K. Claffy. Code-Red: a case study on the spread and victims of an Internet worm. In *Proceedings of the* 2^{nd} *Internet Measurement Workshop (IMW)*, pages 273–284, November 2002.

[30] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[31] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the* 12^{th} *Annual Symposium on Network and Distributed System Security (NDSS)*, February 2005.

[32] M. Owens. Embedding an SQL database with SQLite. *Linux Journal*, 2003(110):2, June 2003.

[33] PaX Home Page. http://pax.grsecurity.net/.

[34] PCWorld. Dangling pointers could be dangerous. http://www.pcworld.com/article/134982/dangling_pointers_could_be_dangerous.html, July 2007.

[35] J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overflows. *IEEE Security & Privacy Magazine*, 2(4):20–27, July/August 2004.

[36] P. Porras, H. Saidi, and V. Yegneswaran. Conficker C analysis. Technical report, SRI International, 2009.

[37] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. In *Proceedings of the* 1^{st} *ACM EuroSys Conference*, April 2006.

[38] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the* 11^{th} *ACM Conference on Computer and Communications Security (CCS)*, pages 298–307, October 2004.

[39] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the* 10^{th} *USENIX Security Symposium*, pages 201–216, August 2001.

[40] A. N. Sovarel, D. Evans, and N. Paul. Where's the FEEB? the effectiveness of instruction set randomization. In *Proceedings of the* 14^{th} *USENIX Security Symposium*, pages 145–160, August 2005.

[41] E. H. Spafford. The Internet worm program: An analysis. Technical Report CSD-TR-823, Purdue University, 1988.

[42] Symantec. Analysis of a zero-day exploit for adobe flash and reader. http://www.symantec.com/connect/blogs/analysis-zero-day-exploit-adobe-flash-and-reader, June 2010.

[43] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, pages 3–17, February 2000.

[44] C. C. Zou, W. Gong, and D. Towsley. Code Red worm propagation modeling and analysis. In *Proceedings of the* 9^{th} *ACM Conference on Computer and Communications Security (CCS)*, pages 138–147, November 2002.

G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries

Kaan Onarlioglu
Bilkent University, Ankara
onarliog@cs.bilkent.edu.tr

Leyla Bilge
Eurecom, Sophia Antipolis
bilge@eurecom.fr

Andrea Lanzi
Eurecom, Sophia Antipolis
lanzi@eurecom.fr

Davide Balzarotti
Eurecom, Sophia Antipolis
balzarotti@eurecom.fr

Engin Kirda
Eurecom, Sophia Antipolis
kirda@eurecom.fr

ABSTRACT

Despite the numerous prevention and protection mechanisms that have been introduced into modern operating systems, the exploitation of memory corruption vulnerabilities still represents a serious threat to the security of software systems and networks. A recent exploitation technique, called Return-Oriented Programming (ROP), has lately attracted a considerable attention from academia. Past research on the topic has mostly focused on refining the original attack technique, or on proposing partial solutions that target only particular variants of the attack.

In this paper, we present G-Free, a compiler-based approach that represents the first practical solution against any possible form of ROP. Our solution is able to eliminate all unaligned free-branch instructions inside a binary executable, and to protect the aligned free-branch instructions to prevent them from being misused by an attacker. We developed a prototype based on our approach, and evaluated it by compiling `GNU libc` and a number of real-world applications. The results of the experiments show that our solution is able to prevent any form of return-oriented programming.

Categories and Subject Descriptors

D.4.6 [**OPERATING SYSTEMS**]: Security and Protection

General Terms

Security

Keywords

Return-oriented programming, ROP, return-to-libc

1. INTRODUCTION

As the popularity of the Internet increases, so does the number of attacks against vulnerable services [3]. A common way to compromise an application is by exploiting memory corruption vulnerabilities to transfer the program execution to a location under the control of the attacker. In these kinds of attacks, the first step requires to find a technique to overwrite a pointer in memory. Overflowing a buffer on the stack [5] or exploiting a format string vulnerability [26] are well-known examples of such techniques. Once the attacker is able to hijack the control flow of the application, the next step is to take control of the program execution to perform some malicious activity. This is typically done by injecting in the process memory a small payload that contains the machine code to perform the desired task.

A wide range of solutions have been proposed to defend against memory corruption attacks, and to increase the complexity of performing these two attack steps [10, 11, 12, 18, 35]. In particular, all modern operating systems support some form of memory protection mechanism to prevent programs from executing code that resides in certain memory regions [33]. The goal of this technique is to protect against code injection attacks by setting the permissions of the memory pages that contain data (such as the stack and the heap of the process) as non-executable.

One of the techniques to bypass non-executable memory without relying on injected code involves reusing the functionality provided by the exploited application. Using this technique, which was originally called return-to-lib(c) [31], an attacker can prepare a fake frame on the stack and then transfer the program execution to the beginning of a library function. Since some popular libraries (such as the `libc`) contain a wide range of functionality, this technique is sufficient to take control of the program (e.g., by exploiting the `system` function to execute `/bin/sh`).

In 2007, Shacham [29] introduced an evolution of return-to-lib(c) techniques [23, 27, 31] called Return-Oriented Programming (ROP). The main contribution of ROP is to show that it is possible for an attacker to execute arbitrary algorithms and achieve Turing completeness without injecting any new code inside the application.

The idea behind ROP is simple: Instead of jumping to the beginning of a library function, the attacker chains together existing sequences of instructions (called Gadgets) that have been previously identified inside existing code. The large availability of gadgets in common libraries allows the attacker to implement the same functionality in many different ways. Thus, removing potentially dangerous functions (e.g., `system`) from common libraries is ineffective against ROP, and does not provide any additional security.

ROP is particularly appealing for rootkit development since it can defeat traditional defense techniques based on kernel data integrity [36] or code verification [24, 28]. Another interesting domain is related to exploiting architectures with immutable memory protection (e.g., to compromise electronic voting machines as shown in [7]). ROP was also recently adopted by real attacks observed in the wild as a way to bypass Windows' Data Execution Prevention (DEP) technology [2].

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

The great interest around ROP quickly evolved into an arms race between researchers. On the one side, the basic attack technique was extended to various processor architectures [6, 7, 14, 15, 34] and the feasibility of mounting this attack at the kernel level was demonstrated [19]. On the other side, ad-hoc detection and protection mechanisms to mitigate the attack were proposed [9, 13, 16, 22]. To date, existing solutions have focused only on the basic attack, by detecting, for instance, the anomalous frequency of return instructions executed [9, 16], or by removing the `ret` opcode to prevent the gadget creation [21]. Unfortunately, a recent advancement in ROP [8] has already raised the bar by adopting different instructions to chain the gadgets together, thus making all existing protection techniques ineffective.

In this paper, we generalize from all the details that are specific to a particular exploitation technique to undermine the foundation on top of which return-oriented programming is built: the availability of instruction sequences that can be reused by an attacker. We present a general approach for the IA-32 instruction set that combines different techniques to eliminate all possible sources of reusable instructions. More precisely, we use code rewriting techniques to remove all unaligned instructions that can be used to link the gadgets. Moreover, we introduce a novel protection technique to prevent the attacker from misusing existing return or indirect jump/call instructions.

We implemented our solution under Linux as a pre-processor for the popular `GNU Assembler`. We then evaluated our tool on different real-world applications, with a special focus on the `GNU libc` (`glibc`) library. Our experiments show that our solution can be applied to complex programs, and it is able to remove all possible gadgets independently from the mechanism used to connect them together. A program compiled with our system is, on average, 26% larger and 3% slower (when all the linked libraries are also compiled with our solution). This is a reasonable overhead that is in line with existing stack protection mechanisms such as StackGuard [11].

This paper makes the following contributions:

- We present a novel approach to prevent an attacker from reusing fragments of existing code as basic blocks to compose malicious functionality.
- To the best of our knowledge, we are the first to propose a general solution to defeat all forms of ROP. That is, our solution can defend against both known variations and future evolutions of the attack.
- We developed *G-Free*, a proof-of-concept implementation to generate programs that are hardened against return-oriented programming. Our solution requires no modification to the application source code, and can also be applied to system applications that contain large sections of assembly code.
- We evaluated our technique by compiling gadget-free versions of `glibc` and other real-world applications.

The rest of the paper is structured as follows: In Section 2, we analyze the key concepts of return-oriented programming.In Section 3, we summarize proposed defense techniques against memory corruption attacks and ROP. In Section 4, we present our approach for compiling gadget-free applications. In Section 5, we describe our prototype implementation. In Section 6, we show the results of the experiments we conducted for evaluating the impact and performance of our system. Finally, in Section 7, we briefly conclude the paper.

2. GADGETS

Before presenting the details of our approach, we establish a more precise and general model for the class of attacks we wish to prevent. Therefore, we generalize the concept of return-oriented programming by abstracting away from all the details that are specific to a particular attack technique.

2.1 Programming with Gadgets

The core idea of return-oriented programming is to "borrow" sequences of instructions from existing code (either inside the application or in the linked libraries) and chaining them together in an order chosen by the attacker. Therefore, in order to use this technique, the attacker has to first identify a collection of useful instruction sequences that she can later reuse as basic blocks to compose the code to be executed. A crucial factor that differentiates return-oriented programming from simpler forms of code reuse (such as traditional return-to-lib(c) attacks) is that the collection of code snippets must provide a comprehensive set of functionalities that allows the attacker to achieve Turing completeness *without* injecting any code [29]. The second step of ROP involves devising a mechanism to manipulate the control flow in order to chain these code snippets together, and build meaningful algorithms.

Note that these two requirements are not independent: To allow the manipulation of the control flow, the instruction sequences must exhibit certain characteristics that impose constraints on the way they are chosen. For example, sequences may have to terminate with a return instruction, or they may have to preserve the content of a certain CPU register. In this paper, we use the term *Gadget* to refer to any valid sequence of instructions that satisfies the control flow requirements.

In a traditional ROP attack, the desired control flow is achieved by placing the addresses of the gadgets on the stack and then exploiting `ret` instructions to fetch and copy them to the instruction pointer. In other words, if we consider each gadget as a monolithic instruction, the stack pointer plays the role of the instruction pointer in a normal program, transferring the control flow from one gadget to the next. Consequently, gadgets are initially defined by Shacham as useful snippets of code that terminate with a `ret` instruction [29].

However, the use of `ret` instructions is just one possible way of chaining gadgets together. In a recent refinement of the technique [8], Checkoway and Shacham propose a variant of ROP in which *return-like* instructions are employed to fetch the addresses from the stack. Because these sequences are quite rare in regular binaries, indirect jumps (e.g., `jmp *%eax`) are used as gadget terminators to jump to a previously identified return-like sequence. In theory, it is even possible to design control flow manipulation techniques that are not stack-based, but that store values in other memory areas accessible at runtime by an attacker (e.g., on the heap or in global variables).

As a result, in order to find a general solution to the ROP threat, we need to identify a property that all possible variants of return-oriented programming have in common. Kornau [34] identified such a property in the fact that every gadget, in order to be reusable, has to end with a "*free-branch*" instruction, i.e., an instruction that can change the program control flow to a destination that is (or that can be under certain circumstances) controlled by the attacker. According to this definition, in each gadget, we can recognize two parts: the *code section* that implements the gadget's functionality and the *linking section* that contains the instructions used to transfer the control to the next gadget. The linking section needs to end with a free branch, but it can also contain additional instructions. For instance, a possible linking section could be the following se-

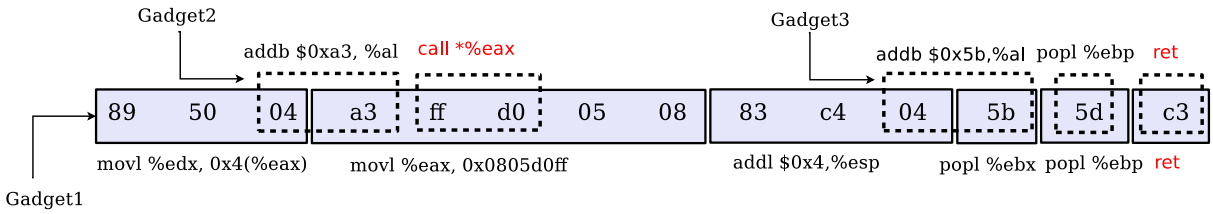


Figure 1: Examples of different gadgets that can be extracted from a real byte sequence

quence: `pop %ebx; call *%ebx`.

2.2 Gadget Construction

In the x86 architecture, gadgets are not limited to sequences of existing instructions. In fact, since the IA-32 instruction set does not have fixed length instructions, the opcode that will be executed depends on the starting point of the execution in memory. Therefore, the attacker can build different gadgets by jumping inside existing instructions.

Figure 1 shows how, depending on the alignment of the first and last instruction, it is possible to construct three different kinds of gadgets. `Gadget1` is an *aligned* gadget that only uses "intended" instructions already present in the function code. `Gadget2` is a gadget that contains only "unaligned" instructions ending with the unintended `call *%eax`. Finally, `Gadget3` starts by using an unintended `add` instruction, then re-synchronizes with the normal execution flow, and ends by reaching the function return. This example demonstrates how a short sequence of 14 bytes can be used for constructing many possible gadgets. Considering that a common library such as `libc` contains almost 18K free branch instructions and that each of them can be used to construct multiple gadgets, it is not difficult for an attacker to find the functionality he needs to execute arbitrary code.

If we can prevent the attacker from finding useful instruction sequences that terminate with a free branch, we can prevent any return-oriented programming technique. We present our approach to reach this goal in Section 4.

3. RELATED WORK

Several defense mechanisms attempt to detect memory exploits which represent a fundamental basic block for mounting return-to-lib(c) attacks. StackGuard [11] and ProPolice [18] are compile-time solutions that aim at detecting stack overflows. PointGuard encrypts pointers stored in memory to prevent them from being corrupted [10]. StackShield [35] and StackGhost [17] use a shadow return address stack to save the return addresses and to check whether they have been tampered with at function exits. A complete survey of traditional mitigation techniques together with their drawbacks is presented in [12]. Our solution, in order to avert ROP attacks, prevents tampering with the return address as well; but it does not target other memory corruption attacks.

One of the most effective techniques that hamper return-to-lib(c) attacks is Address Space Layout Randomization (ASLR) [32]. In its general form, this technique randomizes positions of stack, heap, and code segments together with the base addresses of dynamic libraries inside the address space of a process. Consequently, an attacker is forced to correctly guess the positions where these data structures are located to be able to mount a successful attack. Despite the better protection offered by this mechanism, researchers showed that the limited entropy provided by known ASLR implementations can be evaded either by performing a brute-force attack on 32-bit architectures [30] or by exploiting Global Address Table and de-randomizing the addresses of target functions [25].

Various approaches proposed by the research community aim at impeding ROP attacks by ensuring the integrity of saved return addresses. Frantsen et al. [17] presented a shadow return address stack implemented in hardware for the Atmel AVR microcontroller, which can only be manipulated by `ret` and `call` instructions. ROPdefender [22] uses runtime binary instrumentation to implement a shadow return address stack where saved return addresses are duplicated and later compared with the value in the original stack at function exits. Even though ROPdefender is suitable for impeding basic ROP attacks, it suffers from performance issues due to the fact that the system checks every machine instruction executed by a process.

Another method, called program shepherding [20], can prevent basic forms of ROP as well as code injection by monitoring control flow transfers and ensuring library code is entered from exported interfaces.

Other approaches [9, 13] aim to detect ROP-based attacks relying on the observation that running gadgets results in execution of short instruction sequences that end with frequent `ret` instructions. They proposed to use dynamic binary instrumentation to count the number of instructions executed between two `ret` opcodes. An alert is raised if there are at least three consecutive sequences of five or fewer instructions ending with a `ret`.

The most similar approach to ours is a compiler-based solution developed in parallel to our work by Li et al. [21]. This system eliminates unintended `ret` instructions through code transformations, and instruments all `call` and `ret` instructions to implement return address indirection. Specifically, each call instruction is modified to push onto the stack an index value that points to a return address table entry, instead of the return address itself. Then, when a `ret` instruction is executed, the saved index is used for looking up the return address from the table. Although this system is more efficient compared to the previous defenses, it is presented as a solution specifically tailored for gadgetless kernel compilation, and it exploits characteristics of kernel code for gadget elimination and increased performance. Moreover, the implementation requires manual modifications to all the assembly routines.

It is important to note that none of the defenses proposed so far can address more advanced ROP attacks that utilize free-branch instructions different from `ret`. The solution we present in this paper is the first to address all free-branch instructions, and the first that can be applied at compile-time to protect any program from ROP attacks.

4. CODE WITHOUT GADGETS

Our goal is to provide a proactive solution to build gadget-free executables that cannot be targeted by any possible ROP attack. In particular, we strive to achieve a *comprehensive*, *transparent*, and *safe* solution. By *comprehensive*, we mean that we would like our solution to eliminate all possible gadgets by removing the linking

mechanisms that are necessary to chain instruction sequences together. *Transparent* means that this process must require no intervention from the user, such as manual modifications to the source code. Finally, we would like to present a solution that is *safe*: That is, it should preserve the semantics of the program, be compatible with compiler optimizations, and support applications that contain routines written in assembly language.

In order to reach our goals, we devise a compiler-based approach that first eliminates all unaligned free-branch instructions inside a binary executable, and then protects the aligned free-branch instructions to prevent them from being misused by an attacker.

We achieve the first point through a set of code transformation techniques that ensure free-branch instructions never appear inside any legitimate aligned instruction. This leaves the attacker with the only option of exploiting existing `ret` and `jmp*/call*` instructions. To eliminate this possibility, we introduce a mechanism that protects these potentially dangerous instructions by ensuring that they can be executed only if the functions in which they reside were executed from their entry points.

Consequently, an attacker can only execute entire functions from the start to the end as opposed to running arbitrary code. This effectively de-generalizes the threat to a traditional return-to-lib(c) attack, eliminating the advantages of achieving Turing completeness without injecting any code in the target process.

Our approach uses a combination of techniques, namely alignment sleds, return address encryption, frame cookies and code rewriting. The rest of this section describes each technique in detail.

4.1 Free Branch Protection

The first set of techniques aim to protect the aligned free-branch instructions available in the binary. These include the actual `ret` instructions at the end of each function and the `jmp*/call*` instructions that are sometimes present in the code.

Unfortunately, these instructions cannot be easily eliminated without altering the application's behavior. In addition, replacing them with semantically equivalent pieces of code is likely not going to solve the problem because the attacker could still use the replacements to achieve the same functionality.

Therefore, we propose a simple solution inspired by existing stack protection mechanisms (e.g., StackGuard [11]). The goal is to instrument functions with short blocks of code to ensure that aligned free-branch instructions can only be executed if the running function has been entered from its proper entry point. In particular, we employ two complementary techniques: an efficient return address encryption to protect `ret` instructions, and a more sophisticated cookie-based technique we additionally apply only to those functions that contain `jmp*/call*` instructions. In Section 4.3, we discuss the possibility that an attacker attempts to exploit these protection blocks, and in Section 5.5 we show how we avoid this threat in our prototype.

Finally, we prepend the code performing the checks with *alignment sleds*. Alignment sleds are special sequences of bytes by which we enforce aligned execution of a set of critical instructions. In particular, we use this technique to prevent an attacker from bypassing our free branch protection code by executing it in an unaligned fashion.

4.1.1 Alignment Sleds

An alignment sled is a sufficiently-long sequence of bytes, encoding one or more instructions that have no effect on the status of the execution. Its length is set to ensure that regardless of the alignment prior to reaching the sled, the execution will eventually land on the sled and execute it until the end. Even if an attacker jumps into the binary at an arbitrary point and executes a number of unaligned instructions, when she reaches the sled, the execution will be forced to realign with the actual code. Thus, it will never reach any unintended opcode present in the instructions following the sled.

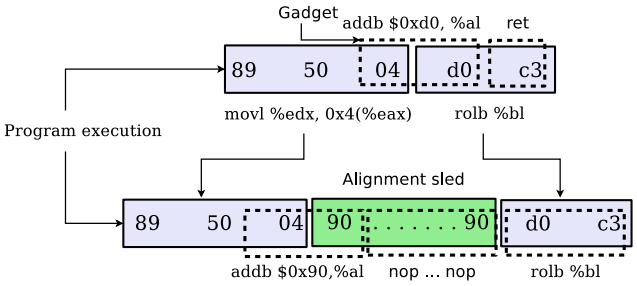


Figure 2: Application of an alignment sled to prevent executing an unaligned `ret` (`0xc3`) instruction

The simplest way to implement an alignment sled is to use a sequence of `nop` instructions (see Figure 2 for an example). The number of `nop` instructions must be determined by taking into consideration the maximum number of consecutive `nop` bytes (`0x90`) that can tail a valid instruction. If we set the length to anything less than that, an attacker could find an unintended instruction that encompasses the whole sled and any number of bytes from the following instruction, in which case the execution will continue in an unaligned fashion. In the IA-32 architecture, the longest such sequence becomes possible when we have both an address displacement and an immediate value entirely composed of `0x90` bytes [4], which makes a total of 8 bytes. Additionally, we can have either a ModR/M byte, a SIB byte or an opcode with the value `0x90` (but only one of them at a time). As a result, we can safely set the number of `nop` instructions in our sled to 9.

Note that the sled length calculation presented in this section is an over-approximation: By also taking into account the bytes preceding the sled and which instructions they can possibly encode, it is possible to automatically compute the required sled length case-by-case.

Finally, we prepend the sled with a relative jump instruction to skip over the sled bytes. Consequently, if the execution is already aligned it will hit the jump and not incur the performance penalty of executing the sequence of `nop` instructions.

4.1.2 Return Address Protection

This technique involves instrumenting entry points of the functions that contain `ret` instructions with a short header that encrypts the saved return address stored on the stack. Before `ret` instructions, we then insert a corresponding footer to restore the return address to its original value. If an attacker jumps into a function at an arbitrary position and eventually reaches our footer, the decryption routine processes the unencrypted return address provided by the attacker, computes an invalid value and the following `ret` instruction attempts to transfer the execution flow to an incorrect address that the attacker cannot control. This technique is similar to the random XOR canary implemented by StackGuard [11].

The encryption method we utilize is a simple exclusive-or of the return address with a *random key*. Since this solution does not affect the layout of the stack in any way, it does not require any further modifications to the function code.

4.1.3 Frame Cookies

In order to prevent the attacker from using existing `jmp*/call*` instructions, we need to adopt another protection mechanism. To

ModR/M	Operand 1	Operand 2
0xc2	%eax, %ax, %al	%edx, %dx, %dl
0xc3	%eax, %ax, %al	%ebx, %bx, %bl
0xca	%ecx, %cx, %cl	%edx, %dx, %dl
0xcb	%ecx, %cx, %cl	%ebx, %bx, %bl

SIB	Base	Scaled Index
0xc2	%edx	%eax*8
0xc3	%ebx	%eax*8
0xca	%edx	%ecx*8
0xcb	%ebx	%ecx*8

Table 1: ModR/M and SIB values encoding `ret` opcodes

this end, we instrument entry points of the functions that contain `jmp*/call*` instructions with an additional header to compute and push a random cookie onto the stack. This cookie is an exclusive-or of a *random key* generated at runtime and a *per-function constant* generated at compile time. The constant is used for uniquely identifying the function and it does not need to be kept secret.

Then, we prepend all the `jmp*/call*` instructions with a *validation block* which fetches the cookie, decrypts it, and compares the result with the *per-function constant*. If the cookie is not found or the values do not match, we invalidate the jump/call destination causing the application to crash. Finally, in the function footer, we insert a simple instruction to remove the cookie from the stack.

A significant consequence of this technique is that it alters the layout of the stack by storing an additional value. This requires us to fix the memory offsets of some of the instructions that access the stack according to the location where we store the cookie (we discuss the details of this issue in Section 5).

4.2 Code Rewriting

The second set of techniques we adopt in our approach focus on removing any unaligned free-branch instructions.

In the IA-32 architecture, instructions consist of some or all of the following fields: instruction prefixes, an opcode, a ModR/M byte, a SIB (Scale-Index-Base) byte, an address displacement, and finally, an immediate value. A `ret` instruction can be encoded with any of the `0xc2`, `0xc3`, `0xca` or `0xcb` bytes, and, as such, can be part of any of the instruction fields (excluding the prefixes). On the other hand, `jmp*/call*` instructions are encoded by two-byte opcodes: an `0xff` followed by an ModR/M byte carrying certain three-bit sequences. Hence, in addition to appearing inside a single instruction, they can also be obtained by a combination of two bytes coming from two consecutive instructions.

In this section, we discuss the various cases and describe the code rewriting techniques we use to eliminate all unintended free-branch opcodes.

4.2.1 Register Reallocation

The ModR/M and the SIB bytes are used for encoding the addressing mode and operands of an instruction. The use of certain registers as operands cause either the ModR/M or the SIB byte to be set to a value that corresponds to a `ret` opcode. The possible undesired encodings of these bytes are shown in Table 1. For instance, an instruction that specifies `%eax` as the source operand and `%ebx` as the destination, such as `movl %eax, %ebx`, assigns the value `0xc3` to the ModR/M byte. Similarly, using `%edx` as the base and `(%ecx * 8)` as the scaled index, the instruction `addl $0x2a,(%edx,%ecx,8)` contains `0xca` in its SIB byte.

In order to eliminate the unintended `ret` opcodes that result from such circumstances, we must avoid all of the undesired register pairings listed in Table 1. We achieve this by manipulating the register allocation performed during compilation to ensure that those pairs of registers never appear together in a generated instruction. When we detect such an instruction, we can perform the compiler's register allocation stage again, this time enforcing a different register assignment. As an alternative, we can perform a local reallocation by temporarily swapping the contents of the original operand with a new register, and then rewriting the instruction with this new register as its operand. In this way, we can bring forth an acceptable register pairing for the same instruction.

Finally, in some cases, the ModR/M byte could be used to specify an opcode extension and a single register operand. Sometimes, it is possible to rewrite these instructions using the same techniques described above to replace the register operand with a different one. However, floating point instructions can use implicit operands that cannot be substituted with others (e.g, `fld %st(2)`). Since all these instructions can have the `ret` opcode only in their second byte, we instead prepend them with an alignment sled. This leaves to the attacker only one byte (the opcode that specifies the FPU instruction) before the unaligned `ret`, and it is therefore impossible to use this byte to create any gadget.

4.2.2 Instruction Transformations

`ret` bytes appear in opcodes encoding `movnti (0x0f 0xc3)` and `bswap (0x0f 0xc8+<register_identifier>)` instructions. In the first case, `movnti` acts like a regular `mov` operation except that it uses a *non-temporal hint* to reduce cache pollution. Thus, we can safely replace it with a regular *mov* without any significant consequence. For the second, the opcode is determined according to the operand register and can encode a `ret` byte when certain registers are specified as the operand; therefore, as described in the previous section, we can perform a register reallocation to choose a different operand and obtain a safe `bswap` opcode.

4.2.3 Jump Offset Adjustments

Jump and call instructions may contain free-branch opcodes when using immediate values to specify their destinations. For instance, `jmp .+0xc8` is encoded as "`0xe9 0xc3 0x00 0x00 0x00`".

A free-branch opcode can appear at any of the four bytes constituting the jump/call target. If the opcode is the least significant byte, it is sufficient to append the forward jump/call with a single `nop` instruction (or prepend it if it is a backwards jump/call) in order to adjust the relative distance between the instruction and its destination:

```
jmp .+0xc8   ⇒   jmp .+0xc9
                 nop
```

However, when the opcode is at a different byte position, the number of `nop` instructions we need to insert increase drastically (256 for the second, 64K for the third and 16M for the last byte).

Fortunately, it is highly uncommon to have a free-branch opcode in one of the most significant bytes. For example, a jump offset encoded by "`0x00 0x00 0xc3 0x00`" indicates a 12MB forward jump. Considering the fact that jump instructions are ordinarily used for local control flow transitions inside a function, a 12MB offset would be infeasible in practice. Even if we were to come across such an offset, it is still possible to relocate the functions or code chunks addressed by the instruction to remove the opcodes.

4.2.4 Immediate and Displacement Reconstructions

Several arithmetic, logic and comparison operations can take immediate values as an operand, which may contain free-branch instruction opcodes. We can remove these by substituting the instruction with a sequence of different instructions that construct the immediate value in steps while carrying the same semantics. The fol-

lowing examples demonstrate the reconstruction process, assuming that %ebx is free or has been saved beforehand:

```
addl $0xc2, %eax   ⇒   addl $0xc1, %eax
                       inc %eax

xorb $0xca, %al    ⇒   movb $0xc9, %bl
                       incb %bl
                       xorb %bl, %al
```

Instructions that perform memory accesses can also contain free-branch instruction opcodes in the displacement values they specify (e.g., `movb %al, -0x36(%ebp)` represented as "0x88 0x45 0xca"). In such cases, we need to substitute the instruction with a semantically equivalent instruction sequence that uses an adjusted displacement value to avoid the undesired bytes. We achieve this by setting the displacement to a safe value and then compensating for our changes by temporarily adjusting the value in the base register. For example, we can perform a reconstruction such as:

```
                                   incl %ebp
movb $0xal, -0x36(%ebp)   ⇒        movb %al, -0x37(%ebp)
                                   decl %ebp
```

4.2.5 *Inter-Instruction Barriers*

Unintended `jmp*/call*` opcodes can result from the combination of two consecutive instructions. This happens when the last byte of an instruction is `0xff` and the first byte of the following instruction encodes a suitable opcode extension. We can remove these unintended `jmp*/call*` opcodes by inserting a *barrier* between two such instructions, effectively separating them and destroying the unintended opcode. For the barrier, the trivial choice of a `nop` instruction is not suitable since an `0xff` followed by a `0x90` still encodes an indirect call. Thus, we have to choose a safe nop-like alternative, such as "`movl %eax, %eax`".

4.3 Limitations of the Approach

By applying the techniques presented in this section, it is possible to remove all unaligned free-branch instructions from a binary, and to protect the aligned ones from being misused by an attacker. However, since our protection mechanism does not remove the free branches, but prepends a short piece of code to protect them, the result of the compilation will still contain some gadgets.

In fact, an attacker may skip the alignment sled by directly jumping *into* the return address or indirect jump/call protection blocks. This may result in executing a useful instruction sequence (intended or unintended) which terminates at the free-branch instruction we intend to protect.

However, since our approach only requires inserting two very short pieces of code, the number of possible gadgets that can be built is very limited and the gadget sizes are restricted to few instructions. By keeping this issue in mind, it is, therefore, possible to specifically craft the return address and indirect jump/call protection blocks to make sure they do not contain any convenient gadgets.

In particular, we discuss the techniques we used in our prototype implementation and the number and type of gadgets that are left in the applications compiled by our tool in Section 5.5.

5. IMPLEMENTATION

Our implementation efforts primarily focus on creating a fully-automated system that would not require any modifications to the program's source code or to the existing compilation tools. Unfortunately, system-wide libraries, which are the primary targets of ROP attacks, often rely on hand-tuned assembly routines to perform low-level tasks. This makes a pure compiler-based solution unable to intercept part of the final code. Therefore, we implemented our prototype in two separate components: an assembly code pre-processor designed to work as a wrapper for the `GNU Assembler (gas)`, and a simple binary analyzer responsible for gathering some information that is not available in the assembly source code.

In this section, we describe G-Free, a prototype system we developed based on the techniques presented in Section 4, and we discuss some of the issues we encountered while compiling `glibc` using our prototype.

5.1 Assembly Code Pre-Processor & Binary Analyzer

The assembly code pre-processor intercepts the assembly code generated by `cc1` (the `GNU C compiler` included in the `GNU Compiler Collection`) or coming directly from an assembly language source file. It then performs the required modifications to remove all the possible gadgets, and finally passes the control to the actual `gas` assembler. We must stress that in this implementation we modify neither the compiler nor the assembler; both are completely oblivious to the existence of our pre-processing stage. We only replace the `gas` executable with a small wrapper responsible for invoking our pre-processor before executing the assembler.

Our system successfully handles assembly routines written using non-standard programming practices. It supports position independent code (PIC) and compiler optimizations, including all of the `GCC` standard optimization levels (in fact, `glibc` *does not* compile if `GCC` optimizations are disabled).

There is one significant implication of directly working with assembly code: Our pre-processor is not exposed to the numeric values of immediate operands and memory displacements since these are often represented by symbolic values until linkage. Thus, it is not possible for us to identify all of the instructions that contain unintended free-branch opcodes just by looking at the assembly code. In order to address this issue, we use a two-step compilation approach. First, our system compiles a given program without doing any modifications to the original code. During this compilation, our pre-processor tags each of the instructions that contain immediate values or displacements with unique symbols. This information is then exported in the final executable's symbol table. In a second step, we use a binary analyzer to read the symbol table of the executable and check whether any of the instructions pointed to by our tagged symbols needs to be rewritten because it contains unaligned free-branch instructions. This analysis produces a log of the tags corresponding to the instructions we need to modify. This log is consumed by the pre-processor during a second compilation phase in order to provide it with the previously missing information.

Unfortunately, inserting a `nop` at a certain position to fix a jump offset may actually affect the offsets of many other jumps since it alters the whole address space of the binary. Our prototype binary analyzer does not consider the overall structure of the binary file when reporting the instructions to fix. Therefore, while fixing a set of jump offsets, several other offsets may start to contain free-branch opcodes. This makes it necessary to perform several compilations until all the offsets are fixed. Note that in this process, we may need to fix a single jump instruction several times. However, since inserting `nop` instructions between a jump and its destination can only increase the offset but never decrease it, we are sure to find a safe offset after a finite number of iterations.

A more optimized analyzer that can perform a global analysis and take into account the target of every jump instruction would eliminate this problem. It would also produce smaller executables since recompilations insert otherwise unnecessary `nop` bytes.

5.2 Random Keys

As described in Section 4, our approach requires a random value to encrypt both the return address and the cookie stored on the stack. For this purpose, our prototype inserts a key generation routine at the beginning of the program's entry point (or initialization routine if it is a library). In our prototype, this routine simply reads a 32-bit random value from the Linux special file */dev/random* and stores the value in a global memory location.

If the attacker has a way to read arbitrary memory locations before performing the actual attack, he could be able to fetch the per-process random key and use it to craft the required values on the stack to defeat our implementation. This limitation is common to many canary-based stack protection mechanism such as StackGuard [11] and ProPolice [18]. However, this problem can be avoided by substituting the per-process random key with a per-function key computed at runtime in the function headers.

5.3 Stack Reference Adjustments

We store our cookie just above the saved return address in the stack, shifting the frame base upwards by 4 bytes. Since a function usually uses the `%ebp` register to reference the stack relative to the frame base, and our cookie is located below the frame base, references to the stack local variables remain unchanged. On the contrary, references to function parameters which are stored below the frame base, and therefore below our cookie, need to be adjusted by 4 bytes.

We achieve this by simply correcting each positive displacement to `%ebp` by adding to it the size of our cookie:

```
movl 0x8(%ebp), %eax   ⇒   movl 0xc(%ebp), %eax
```

Note that compiler optimizations that adopt Frame Pointer Omission (FPO) use the stack pointer to reference arguments and local variables. In this case, we need to compute the displacement of the stack pointer to the function's frame at any given position in the function in order to identify and fix the references and locate our cookie in the stack. This requires a comprehensive stack depth analysis. We have designed our pre-processor to perform this analysis on the fly without the need for any extra pass over the source file, even when the execution flow of the processed function is non-linear. We keep track of `push` & `pop` operations and arithmetic computations on the stack pointer and update the system's view of stack depth accordingly. Depending on the state of the stack, we can then decide whether a stack access (e.g., `120(%esp)`) points to a local variable or to a function's parameter, so that we can apply the displacement adjustment where appropriate.

5.4 Conditional Code Rewriting

Our prototype implements all immediate and displacement reconstruction strategies we described in Section 4. However, to reduce the performance overhead, we apply those transformations only when absolutely necessary. Otherwise, we use a faster approximate solution. In particular, during the first compilation phase, we prepend each instruction that contains free-branch opcodes among its immediate or displacement fields with an alignment sled. The sled protects the instruction, but does not actually remove the free branch from the code. Therefore, an attacker can sometimes build very short gadgets that fit the few bytes between the end of the sled and the unaligned free-branch instruction.

Our system automatically checks these bytes after the compilation. If it detects that they do indeed contain valid instructions, it falls back to the safer (but slightly less efficient) immediate or displacement reconstruction methods.

5.5 Return Address and Indirect Jump/Call Protection Blocks

As previously explained in Section 4, our solution protects aligned free-branch instructions by introducing two short blocks of code: the return address protection block and the indirect jump/call protection block (the current implementations are shown in Figure 3). These two pieces of code are the only ones in the final executable that can still contain gadgets and, therefore, they must be carefully designed to prevent any possible attack.

The return address protection code is 11 bytes long and all bytes are under our control, with the exception of the 4-byte address of the random key, which could change for each compiled program and for shared libraries at each relocation. To ensure that the code is safe to use, we need to prevent this value from containing potentially dangerous instructions. In our implementation, we control the least significant two bytes by automatically inserting appropriate alignment directives into the assembled code when defining the key storage location, ensuring that the address always ends with the innocuous "`0xf0 0x00`" sequence. In addition, according to the Linux process memory layout, the most significant address byte of the `.bss` section (where we store our random key) is limited in practice to `0x08` for regular ELF executables and `0xb*` for shared libraries [1]. Therefore, it encodes either a variation of a *load immediate into register* instruction (e.g., `mov $IMM, %reg`), or an `or` instruction between two 8-bit operands.

The indirect jump/call protection block is 19 bytes long and contains an additional 4-byte-long dynamic section, the per-function constant identifier we generate at compile time to compute the cookie. The example shown in Figure 3 (that uses a `0x0f0f1f76` function identifier) is entirely gadget-free because it contains no aligned or unaligned instruction sequences that would make it possible for an attacker to reach `jmp *%edx` without invalidating its contents. In fact, any logic/arithmetic operation that does not yield a result of zero (e.g., `incl %ebp`, unless `%ebp` overflows) clears the zero flag in the processor and prevents the use of the conditional jump `jz .+4` (this instruction only jumps if the zero flag is set in the processor). Consequently, the value inside `%edx` is cleared.

Different values of the function identifier could potentially introduce a new and useful gadget; but since these constants can be arbitrarily chosen and do not need to be kept secret, we can easily work around problematic cases. In order to minimize the risk in the first place, we use simple heuristics such as using bytes that represent invalid opcodes (e.g., `0x0f 0x0f`) and avoiding dangerous opcodes such as those encoding `mov` or free-branch instructions.

Figure 4 shows all the gadgets that can be extracted from our current system implementation. As can be seen, apart from the ability to load the `%eax` with a controlled value (`popl %eax`), the gadgets have no value.

5.6 Compiling `glibc`

During our case study of compiling `glibc` using G-Free, we have encountered several issues requiring particular care. These were mostly related to unconventional programming practices used for dealing with low-level tasks, or manually optimized assembly code. This section explains our observations in this regard, and explains how we cope with these special cases.

Multiple Entry Points: We have come across various functions in `glibc` that include more than one possible entry point. Our system

[1]The Linux process memory layout dictates that dynamic shared libraries are loaded at the address range `0xc0000000-0x40000000`, starting from higher addresses. As a result, in practice almost any shared library has `0xb*` as the most significant address byte of its `.bss` section.

Return address protection code

```
50              pushl  %eax
a1 00 f0 fd b7  movl   0xb7fdf000, %eax
31 44 24 04     xorl   %eax, 0x4(%esp)
58              popl   %eax
```

Indirect jump/call protection code

```
 50              pushl  %eax
 a1 00 f0 fd b7  movl   0xb7fdf000, %eax
 35 76 1f 0f 0f  xorl   $0x0f0f1f76, %eax
 39 45 04        cmpl   %eax, 0x4(%ebp)
 58              popl   %eax
 74 02           jz     freebranch
 31 d2           xorl   %edx, %edx
freebranch:
 ff e2           jmp    *%edx
```

Figure 3: Code inserted to protect the aligned return and indirect jump/call instructions

Gadget A.1

```
00 f0     addb   %dh, %al
fd        std
b7 31     movb          $0x31, %bh
44        incl          %esp
24 04     andb          $0x04, %al
58        popl          %eax
```

Gadget A.2

```
f0 fd     lock std
b7 31     movb   $0x31, %bh
44        incl   %esp
24 04     andb   $0x04, %al
58        popl   %eax
```

Gadget A.3

```
04 58   addb   $0x58, %al
```

Gadget B.1

```
 45      incl   %ebp
 04 58   addb   $0x58, %al
 74 02   jz     freebranch
 31 d2   xorl   %edx, %edx
freebranch:
 ff e2   jmp    *%edx
```

Figure 4: Gadgets available in the return address (A) and in the indirect jump/call (B) protection blocks

successfully detects such functions and instruments all entry points with the appropriate headers. Additionally, we prepend each header that lies in the execution path of other entry points with a jump instruction to skip over the header, ensuring that only one header is executed per function call.

Functions that Access the Saved Return Address: In `glibc`, we have encountered a single function, namely `setjmp` that accesses the saved return address on the stack. `setjmp`, together with the function `longjmp`, is used for implementing *non-local jumps*: a call to `setjmp` saves the current stack context to restore it afterwards when `longjmp` is invoked. This behavior conflicts with our return address protection scheme. Since the return address is stored in an encrypted form on the stack, a call to `setjmp` saves the encrypted return address and a subsequent call to `longjmp` results in an illegal memory access. In order to solve this problem, we modified our prototype to detect when the saved return address is moved to a register and perform the decryption on the duplicated value to ensure correct functionality.

Jumps between Functions: In numerous cases, a function directly jumps to another one without saving the return address, essentially making that jump an exit point. During compilation, we check every jump destination to recognize jumps outside the current function and treat them as regular exit points for inserting the necessary footers. These footers are not meant to protect a free-branch instruction, since none follows, but to restore the return address to its original value before transferring the execution flow to another function.

6. EVALUATION

The main goal of our evaluation is to show that our technique can be applied to compile real-world applications and produce gadget-free executables. To demonstrate that we are able accomplish this goal, we performed a set of experiments in which we measured the impact of our code transformations in terms of performance and size overheads of the binaries produced by our tool.

In our tests, we combined the G-Free pre-processor with `gas 2.20` and `GCC 4.4.3`. All the experiments were performed on a 2GHz Intel Core 2 Duo T7300 machine with 2GB of memory, running Arch Linux (i686) with Linux kernel 2.6.33.

6.1 Compilation Results

Since ROP attacks usually extract their gadgets from common libraries, we focus our evaluation on `glibc` version 2.11.1. The original version compiled without G-Free contains 9921 `ret` instructions (6106 of which unaligned) and 8018 `jmp*/call*` instructions (6602 of which unaligned). This sums up to almost 18K free-branch opcodes, each of which can be potentially used by an attacker to build many different gadgets.

After we compiled `glibc` using our system, all unintended `ret` and `jmp*/call*` instructions were either removed or made ineffective by prepending them with an alignment sled. In addition, all aligned free-branch instructions were protected by adding our return and indirect jump/call protection blocks. As a result, the library compiled with G-Free contained only the four type of gadgets we present in Figure 4.

However, due to the newly inserted code and instruction rewriting techniques, the size of the gadget-free version of the library increased by 30%. Although this value might appear to be high, one third of the overhead is caused by `nop` instructions included in the alignment sleds. As already discussed in Section 5, most of these could be eliminated by a more optimized implementation.

Unfortunately, providing a gadget-free version of `glibc` is not sufficient to completely prevent ROP attacks, since the attacker could still build the gadget set from other libraries or the application binary itself. Therefore, to achieve a complete protection

Program Name and Version	Original Size(KB)	G-Free Size (Overhead)	Unaligned `ret`	Unaligned `jmp*/call*`	Number of functions with RAP	Number of functions with JCP
glibc 2.11.1	1320.4	1728.4 (30.9%)	6106	6602	2817	827
gzip 1.4	72.7	92.4 (27.0%)	433	410	122	10
grep 2.6.3	86.3	106.3 (23.2%)	523	369	174	20
dd coreutils-8.5	48.0	57.9 (20.6%)	252	181	95	8
md5sum coreutils-8.5	30.9	37.7 (22.1%)	203	86	68	3
ssh-keygen openssh-5.5p1	140.6	182.5 (29.7%)	607	712	271	20
lame 3.98.3	322.6	406.6 (26.0%)	2228	1342	669	28

Table 2: Statistics on binaries compiled with G-Free (RAP=Return Address Protection, JCP=indirect Jump/Call Protection)

Program Name	Test case	Execution Time (seconds) Original Version	G-Free Version (Overhead)
gzip	Compress a 2GB file	66.5	68.4 (2.9%)
grep	Search in a 2GB file	81.3	82.9 (2.0%)
dd	Create a 2GB zero-filled file	86.6	88.9 (2.6%)
md5sum	Compute hash of a 2GB file	82.5	82.9 (0.6%)
ssh-keygen	Generate 100 2048-bit RSA keys	51.2	53.6 (4.6%)
lame	Encode a 10 min long wav file	115.5	122.0 (5.6%)

Table 3: Performance comparisons when the application and all the linked libraries are compiled with G-Free

against ROP, it is necessary to compile the entire application and all the linked libraries with our technique. To demonstrate that our tool can be applied to this more general scenario, we include in our evaluation a number of common Linux applications.

Table 2 shows statistics about the binaries compiled with G-Free. Our tool was able to successfully remove all unintended instructions and protect the aligned ones with an average size increase of 25.9% (more than half of which were caused by redundant `nop` instructions). The last two columns show that most of the functions can be protected by our very efficient return address encryption technique while very few of them required the more complex indirect jump/call protection block. This is a consequence of the fact that, according to what we observed in our experiments, programs rarely use `jmp*/call*` instructions.

6.2 Performance Measurements

Table 3 shows the performance overheads we measured by running the different applications compiled with our prototype (this includes the gadget-free versions of the programs and all their linked libraries). For each application, we designed a set of program-specific test cases, summarized in Column 2 of the table. The average performance overhead was 3.1% – a value comparable with the overhead caused by well known stack protection systems such as StackShield [35] and StackGuard [11].

Since a library cannot be run as a standalone program, we evaluated the performance overhead of our gadget-free version of `glibc` using a set of well-known benchmarks. In particular, we downloaded and installed the Phoronix Test Suite [1] which provides one of the most comprehensive benchmark sets for the Linux platform. Table 4 lists a sample of the benchmarks that represent different application categories such as games, mathematical and physical simulations, 3D rendering, disk and file system activities, compression, and well-known server applications. The results indicate that the performance overhead of an application using our gadget-free version of `glibc` is on average 1.09%.

7. CONCLUSIONS

Return-oriented programming is an attack technique that recently attracted significant attention from the scientific community. Even though much research has been conducted on the topic, no comprehensive defense mechanism has been proposed to date.

In this paper we propose a novel, comprehensive solution to defend against return-oriented programming by removing all gadgets from a program binary at compile-time. Our approach targets all possible free-branch instructions, and, therefore, is independent from the techniques used to link the gadgets together. We implemented our solution in a prototype that we call G-Free, a preprocessor for the `GNU Assembler`. Our experiments show that G-Free is able to remove all gadgets at the cost of a very low performance overhead and an acceptable increase in the file size.

8. ACKNOWLEDGMENTS

The research leading to these results was partially funded from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement nř 257007. This work has also been supported in part by the European Commission through project IST-216026-WOMBAT funded under the 7th framework program. This publication reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein. We would also like to thank Secure Business Austria for their support for this research.

9. REFERENCES

[1] Phoronix test suite. `http://www.phoronix-test-suite.com/`.

[2] Rop attack against data execution prevention technology, 2009. `http://www.h-online.com/security/news/item/Exploit-s-new-technology-trick-%dodges-memory-protection-959253.html`.

[3] Symantec: Internet Security Threat Report. `http://www4.symantec.com/Vrt/wl?tu_id=jLac123913792490340803`, 2009.

[4] Intel 64 and IA-32 Architectures Software Developer's Manuals. `http://www.intel.com/products/processor/manuals/`, 2010.

[5] Aleph One. Smashing the stack for fun and profit. In *Phrack Magazine n.49*, 1996.

[6] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, 2008.

[7] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham. Can DREs Provide Long-Lasting Security? The Case of Return-Oriented Programming and the AVC Advantage.

Benchmark	Orig. Libc	G-Free (Overhead)
FS-Mark (Files/s)	15.1	14.9 (1.3%)
IOzone-write (MB/s)	22.8	22.6 (0.4%)
IOzone-read (MB/s)	23.0	22.7 (1.4%)
Dbench (MB/s)	83.7	82.0 (2.0%)
Minion (s)	250.2	250.7 (0.2%)
Sudokut (s)	97.1	100.4 (3.5%)
TSCP (Nodes/s)	224642.0	224385.0 (0.1%)
GMPbench (Score)	2955.5	2954.5 (0.03%)
BYTE (Lines/s)	7288371.3	6948792.8 (4.6%)
PyBench (s)	6791.0	6959.0 (2.5%)
PHP Comp (s)	102.9	107.3 (4.3%)
7-Zip (MIPS)	2822.0	2802 (0.7%)
Unpack Linux Kernel (s)	30.30	31.01 (2.3%)
LZMA (s)	291.67	291.86 (0.01%)
BZIP2 (s)	65.63	65.84 (0.3%)
FLAC Audio Encoding (s)	12.96	13.09 (1.0%)
Ogg Encoding (s)	27.14	27.20 (0.2%)
Himeno (MFLOPS)	152	151.44 (0.4%)
dcraw (s)	52.68	52.99 (0.6%)
Bullet Physics Engine (s)	39.58	39.74 (0.4%)
Timed MAFFT (s)	52.48	52.55 (0.1%)
PostgreSQL (Trans/s)	155.24	156.66 (0.9%)
SQLite (s)	189.09	191.78 (1.4%)
Apache(Requests/s)	7129.05	6836.24 (4.1%)
x2642009 (Frames/s)	13.72	13.62 (0.7%)
GtkPerf (s)	20.89	20.49 (1.9%)
x11perf (Operations/s)	912000	912000 (0.0%)
Urban Terror (Frames/s)	34.20	34.05 (0.9%)
OpenArena (Frames/s)	46.93	46.67 (0.6%)
C-Ray (s)	553.7	554.0 (0.05%)
FFmpeg (s)	24.93	25.02 (0.4%)
GraphicsMagick (Iter/min)	45	44 (2.2%)
OpenSSL (Signs/s)	25.28	25.28 (0.0%)
Gcrypt Library (micros)	6963	6983 (0.3%)
John The Ripper (Real C/S)	1854667	1857333 (0.1%)
GnuPG (s)	20.46	20.67 (1.0%)
Timed HMMer Search (s)	88.93	89.31 (0.4%)
Bwfirt (s)	284.9	285.3 (0.2%)
Average:		1.09%
Std:		1.27

Table 4: Performance comparison of the original and G-Free `glibc` using benchmarks from the Phoronix Test Suite

In *Proceedings of EVT/WOTE 2009. USENIX/ACCURATE/IAVoSS*, 2009.

[8] S. Checkoway and H. Shacham. Escape from return-oriented programming: Return-oriented programming without returns (on the x86). Technical report, 2010.

[9] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In *Lecture Notes in Computer Science*, 2009.

[10] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities. In *Proceedings of the 12th Usenix Security Symposium,*, 2003.

[11] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7^{th} USENIX Security Symposium, USA*, 1998.

[12] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and Defenses for the Vulnerability of the Decade. In *Proceedings of the DARPA Information Survivability Conference and Exposition,*, 2000.

[13] L. Davi, A. R. Sadeghi, and M. Winandy. Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks. In *Proceedings ACM workshop on Scalable trusted computing*, 2009.

[14] Felix Lidner. Confidence 2.0,. Developments in Cisco IOS forensics.

[15] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of CCS*, 2008.

[16] A. Francillon, D. Perito, and C. Castelluccia. Defending embedded systems against control flow attacks. In *Proceedings of the first ACM workshop on Secure execution of untrusted code*, 2008.

[17] M. Frantsen and M. Shuey. Stackghost: Hardware facilitated stack protection. In *Proceedings of USENIX security*, 2001.

[18] Hiroaki Etoh. GCC Extension for Protecting Applications from Stack-Smashing Attacks (ProPolice). In *http://www.trl.ibm.com/projects/security/ssp/*, 2003.

[19] R. Hund, T. Holz, and F. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Proceedings of the 18th USENIX Security Symposium, USA*, 2009.

[20] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[21] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating Return-Oriented Rootkits with Return-less Kernels. In *Proceedings of the 5th ACM SIGOPS EuroSys Conference*, 2010.

[22] M. W. Lucas Davi, Ahmad-Reza Sadeghi. Ropdefender: A detection tool to defend against return-oriented programming attacks. Technical report, Technical Report HGI-TR-2010-001.

[23] Nergal. The advanced return-into-lib(c) exploits. In *Phrack Magazine n.58*, 2001.

[24] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 1–20, Berlin, Heidelberg, 2008. Springer-Verlag.

[25] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the 25^{th} Annual Computer Security Applications Conference (ACSAC), Honolulu, Hawaii, USA.*, pages 60–69. IEEE Computer Society, Dec. 2009.

[26] Scut, Team Teso. Exploiting format string vulnerabilities. 2001.

[27] Sebastian Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique, 2005. `http://www.suse.de/~krahmer/no-nx.pdf`.

[28] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of Operating System Symposium SOSP*, 2007.

[29] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, 2007.

[30] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CSS)*, 2004.

[31] Solar Designer. return-to-libc attack. Technical report, bugtraq, 1997.

[32] The PaX Team. Pax address space layout randomization. Technical report, http://pax.grsecurity.net/docs/aslr.txt.

[33] The PaX Team. Pax non-executable pages. Technical report, http://pax.grsecurity.net/docs/noexec.txt.

[34] Tim Kornau. Return oriented programming for the arm architecture. Technical report, Master's thesis, Ruhr-Universität Bochum, 2010.

[35] Vendicator. Stackshield: A "stack smashing" technique protection tool for linux. Technical report, http://www.angelfire.com/sk/stackshield/.

[36] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS*, 2009.

Towards Practical Anonymous Password Authentication

Yanjiang Yang, Jianying Zhou, Jun Wen Wong, Feng Bao
Institute for Infocomm Research
Singapore 138632
{yyang,jyzhou,jwwong,baofeng}@i2r.a-star.edu.sg

ABSTRACT

The conventional approach for anonymous password authentication incurs $\mathcal{O}(N)$ server computation, linear to the total number of users. In ACSAC'09, Yang *et al.* proposed a new approach for anonymous password authentication, breaking this lower bound. However, Yang *et al.*'s scheme has not considered *membership withdrawal* and *online guessing attacks*, two issues must be addressed before anonymous password authentication is acceptable for practical use. Thus our main thrust in this work is to provide solutions to these issues. We do not just work upon Yang *et al.*'s scheme; rather, we use a set of different primitives, and as a result, our scheme has much better performance. We prove the security of our scheme. Furthermore, we empirically evaluate the efficiency of our scheme, and implement a proof-of-concept prototype.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Security

Keywords

password authentication, anonymity, guessing attack

1. INTRODUCTION

To meet the growing need for protection of individual privacy, recently anonymous password authentication has been proposed to reinforce regular password authentication with additional protection of user privacy, such that logins made by the same user cannot be linked by the server. An inherent drawback of anonymous password authentication in the conventional setting, where the server keeps a password file containing N user passwords, is that the server has to perform $\mathcal{O}(N)$ computation in responding to a login request. This clearly makes the system unscalable when the number of registered users is large. To get over this barrier, in ACSAC'09 Yang *et al.* [36] proposed a new approach to achieve anonymous password authentication.

At the core of Yang *et al.*'s approach stands the concept of password-protected credentials. In particular, a user is issued a credential to be used for authentication, and she protects the credential with a password; subsequently to log-in with the server, the user first releases the credential from her password-protected credential using the password, and then engages in the authentication process with the server using the credential. With this approach, the cost incurred upon the server is to check the validity of the user's credential, thus constant. It can be seen that Yang *et al.*'s approach actually trades portability of password for server efficiency. Thus for the approach to be useful, the password-protected credentials must not require any *secure facility* for storage, and they can be entrusted to any portable devices, even public directories.

Yang *et al.*'s approach indeed makes a step forward over the conventional approach for anonymous password authentication [36]. However, that scheme is still far from being of practical use. (1) It has no support of *membership withdrawal*. In practical applications, from time to time the server often needs to revoke some users' access right under certain circumstances. (2) It does not consider *online guessing attacks*. In fact, none of the other existing anonymous password authentication schemes [1, 31, 34, 35] has ever considered this issue. Indeed, it turns out that online guessing attacks become particularly troublesome in the setting of anonymous password authentication. On the one hand, the server needs to discern users in case of attacks, so as to take system-level measures as in regular password authentication, e.g., block the victim user's account. On the other hand, discrimination of users is not allowed in order to preserve user privacy, and worse yet, there does not always exist a TTP (Trusted Third Party) to help revoke user anonymity. This contrasts sharply to other privacy preserving primitives (e.g., group signature and anonymous credential) where a TTP is often assumed for anonymity revocation.

Our Contributions. To advance anonymous password authentication a step further towards practicality, in this work we attempt to solve the above problems existed in Yang *et al.*'s scheme. Specifically, our contributions are summarized below.

- We adopt the BBS signature [5, 2] (in place of the CL signature [16] in Yang *et al.*'s scheme) to establish a basic scheme, which is comparable to Yang *et al.*'s scheme. Our scheme has considerably better efficiency in terms of both

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

computation and communication.

- We then extend the basic scheme to support membership withdrawal, by employing the dynamic accumulator [24]. We stress that while the idea of using dynamic accumulator to facilitate membership withdrawal is not a surprise, considering the fact that such a tool has been commonly used in group signature and alike, we must be very careful in handling the witnesses (resulted from the accumulator) amid the password protected credentials, so as not to inflict offline guessing attacks.

- We are the first to consider online guessing attacks in anonymous password authentication. In fact, addressing online guessing attacks in this setting poses tremendous challenges. In particular, we discuss various methods, and our final countermeasure is a *virtual* TTP (vTTP) based anonymity revocation strategy, where the server and a number of (e.g., t-out-of-N) users form a virtual TTP, such that TTP-based anonymity revocation is enabled.

- We formulate and analyze the security of our scheme. Furthermore, we empirically evaluate the performance of our scheme, and obtain promising experimental results. We also implement a simple proof-of-concept prototype to illustrate the usage of our scheme in practice.

Organization. The rest of the work is organized as follows. In Section 2, we review the related work, followed by Section 3 which provides an overview of the main cryptographic primitives to be used in our constructions. In Section 4, we propose a basic scheme using the BBS signature, which is then extended to support membership withdrawal. We discuss to address online guessing attacks in Section 5. Security definition and analysis of our scheme are presented in Section 6. Section 7 reports the results of experimental performance evaluation and proof-of-concept prototype implementation, and Section 8 concludes the work.

2. RELATED WORK

Password authentication is an intensively explored field in the literature, e.g., [7, 8, 10, 11, 21, 23, 26]. It is well known that the best a password authentication protocol can achieve (in terms of the security of user passwords) is that an attacker has no way to validate his guesses of password unless interacts online with the server and sees the server's accept/reject feedbacks. Such online guessing attacks are inherent because of the low entropy of passwords, and can only be addressed by complementary system-level measures, e.g., locking one's account once the number of failed login attempts made in the name of a user exceeds a threshold.

It is also clear that in general, password authentication does not protect user privacy. Anonymous password authentication is a primitive strengthening password authentication with the protection of user privacy. The first anonymous password authentication scheme was proposed in [34], which combines a password-only protocol with a PIR (Private Information Retrieval) protocol, with the former generating a shared key between the user and the server, and the latter achieving user privacy protection. Afterwards, several schemes that improve [34] in one way or another were presented [31, 35]. In addition, [1] considered anonymous password authentication in a three-party scenario (i.e., user-gateway-server). The server computation in all of these schemes [1, 34, 31, 35] is $\mathcal{O}(N)$, linear to the total number of N users. This is not surprising, and it can be shown that $\mathcal{O}(N)$ is the lower bound in achieving anonymous password authentication in the conventional setting where the server holds a password file containing all user passwords. To break this bound, in ACSAC'09 Yang *et al.* [36] proposed a new approach of password-protected authentication credentials. However, Yang *et al.*'s scheme does not support membership withdrawal and does not consider online guessing attacks, two issues must be addressed for anonymous password authentication to be useful in reality.

We all know that safe management of private keys is fundamental to the use of public key primitives. Although smartcards can be used to store private keys, it is not convenient, as smartcards need the supporting infrastructure (e.g, smartcard reader) to operate. The concept of *password-enabled PKI* [30] has been proposed to solve this issue, with the private keys being protected by passwords. There are two general approaches for password-enabled PKI: (1) to store a private key on a trusted storage facility, and when needed, the owner retrieves the private key after authenticating to the facility using password [27, 32]; (2) to split a private key into two parts, where the owner holds a part derived from her password, and a trusted facility holds the other part; a use of the private key requires the two parts to cooperate. A concern of both approaches is that the storage facility must be trusted, as it learns the private keys.

The *software smartcard* [22] can be viewed as a special case of password-enabled PKI, without requiring the presence of a trusted facility. Its main idea is to encrypt private keys with passwords, and an encrypted private key does not need further protection. For such password-encrypted private keys to be secure against offline guessing attacks, the corresponding public keys must be hidden; otherwise, the relationship between public/private keys allows for offline guessing attacks. This, however, contradicts the main advantage of PKI that the public keys are public. As a matter of fact, the situation faced by password-encrypted private keys is quite similar to that confronted by password-protected credentials. However, as the credentials are to be used upon a server only, it is not an issue to hide the verifiability of credentials from other subjects.

In [4], an approach was proposed on how to store/retrieve a password-scrambled credential on/from a storage facility, while without letting the facility learn the password/credential. This actually amounts to the very first element of our whole approach, namely, a password-protected credential can be placed to any facility. It is simply assumed in [4] that no external information (e.g., from the protocol where a credential is used) is available to assist offline guessing attacks, while we are going far beyond that by looking into the harder situation of concretizing this assumption.

3. PRELIMINARIES

3.1 Notations

Bilinear Pairings. Let G_1, G_2, G_T be cyclic groups of prime order q. A bilinear map $e : G_1 \times G_2 \rightarrow G_T$ has the following properties.

- Bilinear: $\forall u \in G_1, v \in G_2$ and $x, y \in_R Z_q$, $e(u^x, v^y) = e(u, v)^{xy}$. Throughout the paper, $a \in_R S$ means a is randomly chosen from S.
- Non-degenerate: let g be a generator of G_1, and h be a generator of G_2, $e(g, h) \neq 1$.

3.2 Building Primitives

BBS Signature. Boneh *et al.* [5] proposed short group signature (BBS signature for short), and later [2] transferred the BBS signature into a signature with efficient protocols, like [16]. Let $e : G_1 \times G_2 \rightarrow G_T$ be a bilinear map as defined above, and h be a generator of G_2. The public key of the BBS signature is $(W = h^{\chi}, h \in G_2, a, b, d \in G_1)$, and the private key is $(\chi \in Z_q)$. A BBS signature signed upon a message m is defined to be (M, k, s), where $k, s \in_R Z_q$, and $M = (a^m \cdot b^s \cdot d)^{\frac{1}{k+\chi}} \in G_1$.

Signature Verification: A BBS signature (M, k, s, m) can be verified with respect to the public key as $e(M, W \cdot h^k) = e(a, h)^m \cdot e(b, h)^s \cdot e(d, h)$. This verification procedure can be carried out in a zero-knowledge proof protocol that enables the holder of the signature to prove the possession of (M, k, s, m) to a verifier, while without revealing any information on the signature. For brevity, we denote the zero-knowledge proof by $PoK\{(M, k, s, m) : e(M, W \cdot h^k) = e(a, h)^m \cdot e(b, h)^s \cdot e(d, h)\}$, and the details can be found in [2].

Dynamic Accumulator. Dynamic accumulator is a primitive allowing a large set of values to be accumulated into a single quantity, the *accumulator*; and for each value, there exists a *witness*, which is the evidence attesting that the value is indeed contained in the accumulator. The proof of showing that a value is accumulated in an accumulator can be zero-knowledge, which reveals nothing to the verifier on the value and the witness. A concrete construction of dynamic accumulator is due to Nguyen [24], with the details as follows. Let $e : G_1 \times G_2 \rightarrow G_T$ be a bilinear map, and $\bar{g}, \hbar$ be generators of G_1, G_2, respectively. The public parameters include $(W_{acc} = \hbar^{\chi_{acc}}, \hbar)$, and the private key is $(\chi_{acc} \in Z_q)$. The accumulator for a set of values $(v_1, v_2, \cdots, v_\ell)$ is defined as $\Lambda = \bar{g}^{\prod_{j=1}^{\ell}(v_j + \chi_{acc})}$. For v_i, the witness is $w_i = \bar{g}^{\prod_{j=1, j\neq i}^{\ell}(v_j + \chi_{acc})}$. As such, (w_i, v_i) satisfies $w_i^{v_i + \chi_{acc}} = \Lambda$, which can be verified by $e(w_i, W_{acc} \cdot \hbar^{v_i}) = e(\Lambda, \hbar)$ using the public information. For brevity, the zero-knowledge proof showing that v_i is accumulated in Λ is denoted by $\Pi_{acc} = PoK\{(w_i, v_i) : e(w_i, W_{acc} \cdot \hbar^{v_i}) = e(\Lambda, \hbar)\}$, with the details in [2, 24].

4. A SCHEME WITH SUPPORT OF MEMBERSHIP WITHDRAWAL

In this section, we first present a basic scheme, which is comparable to Yang *et al.*'s scheme in [36]. We then extend the scheme to support membership withdrawal.

4.1 Overview

Recall that in Yang *et al.*'s scheme, a password-protected credential essentially consists of two parts: one is a symmetric encryption of certain elements of the user credential with password, while the other is an encryption of other elements under the server's homomorphic encryption. The password-protected credential does not solicit further protection, as the homomorphic encryption part has broken the *verifiable* structure of the credential with respect to outsiders, and achieves *limited verifiability* to the server only. Moreover, to withstand offline guessing attacks, the elements encrypted by password themselves must not have recognizable patterns, and should be uniformly random in the domains where they are drawn[1]. In Yang *et al.*'s scheme, the CL signature [16] is used to instantiate user credentials, and the Paillier encryption [28] is used to implement the server's homomorphic encryption. This combination actually yields a very expensive scheme: the Paillier encryption works in a group of $Z_{n^2}^*$ (where n is a RSA-type modulus), while the CL signature involves costly zero-knowledge range proofs , e.g., [3].

Aimed at better performance, we choose the BBS signature [5, 2] in our construction to replace the CL signature. On the one hand, the BBS signature does not invoke any range proof; on the other hand, this allows us to evade the Paillier encryption and to use more efficient multiplicative homomorphic encryption (e.g., the ElGamal encryption). Better yet, to achieve the same level of security, the required length of the group elements in G_1 of the BBS signature is much shorter than that in the RSA-type group of the CL signature, which results in much shorter protocol transcript in our scheme. For concreteness, a performance comparison is provided shortly. In particular, a user's credential is defined to be a BBS signature (M, k, s) signed on the user's identity u, subject to $M = (a^u \cdot b^s \cdot d)^{\frac{1}{k+\chi}}$ (see above).

R-BBS Signature. We still follow Yang *et al.*'s rationale in achieving limited verifiability: the server holds homomorphic encryption, and users partially encrypt their credentials to break the public verifiability of the credentials. But the homomorphic encryption used here is multiplicative, satisfying $\mathtt{E}(m_1) \cdot \mathtt{E}(m_2) = \mathtt{E}(m_1 \cdot m_2)$, where $\mathtt{E}(.)/\mathtt{D}(.)$ denote encryption/decryption, respectively. The reason for multiplicative homomorphic encryption is that there exist efficient schemes, e.g., the standard ElGamal encryption.

In particular, we choose to encrypt s, and a user's password-protected credential is $\langle u, [M, k]_{pw}, \mathtt{E}(s)\rangle$. To avoid linkage by the server, each time in using the credential, the user first randomizes $\mathtt{E}(s)$ by computing $s^* = \mathtt{E}(r) \cdot \mathtt{E}(s) = \mathtt{E}(r \cdot s)$, where $r \in_R Z_q$. Then the user sends s^* to the server, who decrypts and computes $B = b^{r \cdot s}$. Since the user does not know $r.s$, she cannot perform $PoK\{(M', k, u', s', r) : e(M', W \cdot h^k) = e(a, h)^{u'} \cdot e(b, h)^{s'} \cdot e(d, h)^r\}$, where $M' = M^r, u' = r \cdot u, s' = r \cdot s$. Fortunately, since it holds that $e(M, W \cdot h^k) = e(a, h)^u \cdot e(B, h)^{r^{-1}} \cdot e(d, h)$, the user can perform $PoK\{(M, k, \gamma, u) : e(M, W \cdot h^k) = e(a, h)^u \cdot e(B, h)^{\gamma} \cdot e(d, h)\}$, where $\gamma = r^{-1} \pmod q$ (Note that although the user does not know $r \cdot s$, she can still compute $e(B, h)$ from the equation). This proof is essentially a variant of the original BBS signature, by proving a *randomization* of the original signature (M, k, s, u). We thus refer to it as R-BBS signature. The reason why the server accepts this proof is due to the obvious reduction from R-BBS signature to BBS signature: a forger that produces a R-BBS signature (M, k, γ, u) and $r \cdot s$ clearly generates a BBS signature $(M, k, r \cdot s \cdot \gamma, u)$.

Instantiation of R-BBS Signature. To be specific, below we give an instantiation of R-BBS, denoted by $\Pi_{\textit{R-BBS}}$. Recall that the prover has in possession M, k, γ, and $e(B, h)$. The prover begins by committing to M as $T_1 = M \cdot g_0^{\alpha}, T_2 = g_1^{\alpha}$, where $g_0, g_1 \in G_1$ are pre-defined parameters, and $\alpha \in_R Z_q$.

[1]In Yang *et al.*'s scheme, the elements encrypted by password are v, k, s_2. While Yang *et al.* did not specify how the encryption proceeds, it should be clear that direct encryption of k is not secure, as k in the CL signature [16] is of a particular length. Hence, k must be left in the clear amid the password-protected credential in their scheme.

Then, the following holds:

$$\begin{aligned} e(T_1, W \cdot h^k) &= e(T_1, W) \cdot e(T_1, h)^k \\ &= e(M, W \cdot h^k) \cdot e(g_0^\alpha, W \cdot h^k) \\ &= e(a,h)^u \cdot e(B,h)^\gamma \cdot e(d,h) \cdot e(g_0, W)^\alpha \cdot \\ &\quad e(g_0,h)^{\alpha \cdot k} \end{aligned} \quad (1)$$

Let $\tilde{\alpha} = \alpha \cdot k$. From Equation (1), the user clearly needs to prove the knowledge of $u, k, \gamma, \alpha, \tilde{\alpha}$, subject to the following relations:

$$\begin{aligned} \frac{e(T_1,W)}{e(d,h)} &= \left(\frac{1}{e(T_1,h)}\right)^k \cdot e(a,h)^u \cdot e(B,h)^\gamma \cdot e(g_0,W)^\alpha \cdot \\ &\quad e(g_0,h)^{\tilde{\alpha}}, \\ T_2 &= g_1^\alpha, \quad 1 = (\frac{1}{T_2})^k \cdot g_1^{\tilde{\alpha}} \end{aligned} \quad (2)$$

The proof of knowledge of $u, k, \gamma, \alpha, \tilde{\alpha}$ proceeds as follows. The user picks random $r_u, r_k, r_\gamma, r_\alpha, r_{\tilde{\alpha}} \in_R Z_q$, and computes the following:

$$\begin{aligned} R_1 &= \left(\frac{1}{e(T_1,h)}\right)^{r_k} \cdot e(a,h)^{r_u} \cdot e(B,h)^{r_\gamma} \cdot e(g_0,W)^{r_\alpha} \cdot \\ &\quad e(g_0,h)^{r_{\tilde{\alpha}}}, \\ R_2 &= g_1^{r_\alpha}, \quad R_3 = (\frac{1}{T_2})^{r_k} \cdot g_1^{r_{\tilde{\alpha}}} \end{aligned} \quad (3)$$

We define $\{T_1, T_2, R_1, R_2, R_3\} = \mathtt{Cmt}(\Pi_{R\text{-}BBS})$. The prover sends $\mathtt{Cmt}(\Pi_{R\text{-}BBS})$ to the verifier, who sends back a random challenge $c \in Z_q$. Upon receipt of the challenge, the prover continues to compute (modulo q):

$$\begin{aligned} s_u = r_u + c \cdot u \quad s_\gamma = r_\gamma + c \cdot \gamma \quad s_k = r_k + c \cdot k \\ s_\alpha = r_\alpha + c \cdot \alpha \quad s_{\tilde{\alpha}} = r_{\tilde{\alpha}} + c \cdot \tilde{\alpha} \end{aligned} \quad (4)$$

Let $\{s_u, s_\gamma, s_k, s_\alpha, s_{\tilde{\alpha}}\} = \mathtt{Res}(\Pi_{R\text{-}BBS})$. The prover sends $\mathtt{Res}(\Pi_{R\text{-}BBS})$ to the verifier, who accepts if all the following hold:

$$R_2 \cdot T_2^c = g_1^{s_\alpha} \quad (5)$$

$$R_3 = (\frac{1}{T_2})^{s_k} \cdot g_1^{s_{\tilde{\alpha}}} \quad (6)$$

$$\begin{aligned} R_1 \cdot \left(\frac{e(T_1,W)}{e(d,h)}\right)^c &= \left(\frac{1}{e(T_1,h)}\right)^{s_k} \cdot e(a,h)^{s_u} \cdot e(B,h)^{s_\gamma} \cdot \\ &\quad e(g_0,W)^{s_\alpha} \cdot e(g_0,h)^{s_{\tilde{\alpha}}} \end{aligned} \quad (7)$$

For this instantiation, we have the following theorem, and the proof is not included herein due to the limited space.

THEOREM 1. *The above instantiation, $\Pi_{R\text{-}BBS}$, is honest-verifier zero-knowledge proof of knowledge of a tuple (M, k, γ, u), subject to*

$$e(M, W \cdot h^k) = e(a,h)^u \cdot e(B,h)^\gamma \cdot e(d,h)$$

4.2 Details of Basic Scheme

Based on the above discussions, we present a basic scheme as follows.

Setup: To set up the system parameters, the server does the following:

(A) determines a bilinear map $e : G_1 \times G_2 \to G_T$ as defined earlier, and sets up the public key for the BBS signature as $(W = h^\chi \in G_2, h \in G_2, a, b, d \in G_1)$, and the private key as $(\chi \in Z_q)$;

(B) publishes $g \in G_1$ as part of the public information;

(C) selects a public/private key pair for ElGamal encryption, with $\mathtt{E}(.)/\mathtt{D}(.)$ denoting encryption/decryption, respectively;

(D) picks a hash function $H : \{0,1\}^* \to \{0,1\}^{\kappa_0}$, and a MAC $\mathtt{MAC} : \{0,1\}^{\kappa_1} \times G_1^2 \to \{0,1\}^{\kappa_1}$, where κ_0, κ_1 are appropriate numbers.

Registration: Users need to register to the server in advance, each getting an authentication credential. The server issues each user a credential, which is a BBS signature (M, k, s) signed upon the user identity u. Upon receipt of her credential, the user protects (M, k) using a symmetric key encryption with a key derived from her password pw, i.e, $\widehat{cred}_1 = [M, k]_{pw}$; encrypts s using the server's public key, i.e., $\widehat{cred}_2 = \mathtt{E}(s)$. The complete password-protected credential is $\langle u, \widehat{cred}_1, \widehat{cred}_2 \rangle$. Finally, the user puts the password-protected credential to her preferred storage, e.g., handphone, USB flash memory, or a public facility.

Authentication Protocol: Suppose a user u already has her password-protected credential $\langle u, \widehat{cred}_1, \widehat{cred}_2 \rangle$ available at the point of login. The authentication protocol between the user and the server is as follows.

Step 1. The user does the following:

(1) recovers (M, k) by opening $\widehat{cred}_1$ with her password pw;
(2) picks random $r \in_R Z_q$ to randomize $\widehat{cred}_2$ by computing $s^* = \mathtt{E}(r) \cdot \mathtt{E}(s) = \mathtt{E}(r \cdot s)$;
(3) picks $x \in_R Z_q$, and computes $X = g^x$;
(4) picks $N_A \in_R \{0,1\}^{\kappa_1}$, and computes $N_A^* = \mathtt{E}(N_A)$.
(5) constructs $\mathtt{Cmt}(\Pi_{R\text{-}BBS})$ using the above R-BBS signature instantiation over $(M, k, \gamma = r^{-1} \pmod q, u)$; Finally sends $s^*, X, N_A^*, \mathtt{Cmt}(\Pi_{R\text{-}BBS})$ to the server as a login request:

$$\mathsf{User} \longrightarrow \mathsf{Server}: \ s^*, X, N_A^*, \mathtt{Cmt}(\Pi_{R\text{-}BBS})$$

Step 2. Upon receipt of the login request, the server does the following:

(1) computes, in turn, $s' = \mathtt{D}(s^*)$, and $B = b^{s'}$;
(2) picks $y \in_R Z_q$ and computes $Y = g^y$;
(3) computes $N_A' = \mathtt{D}(N_A^*)$, and computes $\mathtt{Mac} = \mathtt{MAC}(N_A', Y, X)$;
(4) picks $N_B \in_R Z_q$, and sends back $N_B, Y, \mathtt{Mac}$ to the user:

$$\mathsf{Server} \longrightarrow \mathsf{User}: \ N_B, Y, \mathtt{Mac}$$

Step 3. The user does the following:

(1) validates $\mathtt{Mac}$, and aborts if invalid;
(2) taking N_B as challenge, constructs and sends $\mathtt{Res}(\Pi_{R\text{-}BBS})$ to the server;
(3) ends the protocol by computing a shared key $sk = H(N_A, N_B, Y^x)$.

$$\mathsf{User} \longrightarrow \mathsf{Server}: \ \mathtt{Res}(\Pi_{R\text{-}BBS})$$

Step 4. The server computes $sk = H(N_A, N_B, X^y)$ upon verification of $\mathtt{Res}(\Pi_{R\text{-}BBS})$.

It is not necessary to elaborate on the protocol step by step, as the description is already clear. We only point out that the user authenticates to the server by showing the possession of a valid credential, while authentication of the server to the user is by the fact that only the server can correctly decrypt N_A^*.

Performance Comparison. To be specific on the performance advantages of our scheme, below we report an analytic per-

Table 1: Performance Comparison

	Computation[a]		Communication (bits)
	User	**Server**	
Yang *et al.*'s scheme	$48 \cdot \texttt{EXP}_N + 4 \cdot \texttt{EXP}_{N^2}$	$55 \cdot \texttt{EXP}_N + 2 \cdot \texttt{EXP}_{N^2}$	$33\|G_N\| + 2\|G_{N^2}\|$
Our Scheme	$11 \cdot \texttt{EXP}_{G_1} + 5 \cdot \texttt{EXP}_{G_T}$ $+2 \cdot \texttt{PAIRING}$[b]	$7 \cdot \texttt{EXP}_{G_1} + 6 \cdot \texttt{EXP}_{G_T}$ $+4 \cdot \texttt{PAIRING}$[b]	$7\|G_1\| + \|G_T\| + 6\|q\|$

[a] We mainly count # of EXPonentiations; besides, a multi-exponentiation is treated as multiple EXPs (e.g., $g_0^x g_1^y$ is counted as 2 EXPs).
[b] We treat $e(a,h), e(d,h), e(g_0,h)$ as part of the public system parameters, as they are fixed quantities.

formance comparison between our scheme and Yang *et al.*'s scheme. For fairness, let the two schemes achieve the same level of security, e.g., 80 bits, then $|q| = 160$ bits, $|G_1| = 171$, $|G_2| = |G_T| = |G_N| = 1024$ bits (see, e.g., [20]); moreover, the following relation holds for computation cost, $1 \cdot \texttt{EXP}_{G_1} \approx 1.\texttt{EXP}_{G_T} \approx 1.\texttt{EXP}_{G_N} \approx 0.25 \cdot \texttt{EXP}_{G_{N^2}} \approx 0.25 \cdot \texttt{PAIRING}$. As such, it can easily see that our scheme is considerably superior to Yang *et al.*'s scheme in terms of both computation and communication, according to Table 1.

4.3 Support of Membership Withdrawal

We next extend the above scheme to support membership withdrawal. Once a user withdraws, her credential must be revoked by the server. To this end, we employ the dynamic accumulator in [24]. In particular, the server accumulates k's of all valid credentials (recall that a user's credential is (M, k, s)), and each user is issued a corresponding witness; in authentication, a user needs to show that the k element of her credential is contained in the server's accumulator with the help of the witness. We stress that while the idea of using dynamic accumulator to accommodate membership withdrawal is straightforward, care must be taken in handling the witness amid the password-protected credential, so as not to incur offline guessing attacks. Below we outline the extensions to be made to the above basic scheme.

In the **Setup** phase, the server also sets up public key $(W_{acc} = \hbar^{\chi_{acc}}, \hbar \in G_2)$ and private key $(\chi_{acc} \in Z_q)$ for the dynamic accumulator, under the same bilinear map $e : G_1 \times G_2 \rightarrow G_T$ of the BBS signature. In the **Registration** phase, the server accumulates k's of all users, and publishes an accumulator $\Lambda = \bar{g}^{\prod_{j=1}^{\ell}(k_j + \chi_{acc})}$ at the end of the registration phase. Accordingly, a user u is given a witness w, along with the credential (M, k, s). We must be very careful in protecting w amid the password protected credential. Specifically, since w, k form a *verifiable* pair, i.e., $e(w, W_{acc} \cdot \hbar^k) = e(\Lambda, \hbar)$, neither of them can be protected by password; otherwise, offline guessing attacks apply. Hence both will be left in the clear. Thus a user's password-protected credential is $\langle u, k, w, \widehat{cred}_1 = [M]_{pw}, \widehat{cred}_2 = \texttt{E}(s)\rangle$. Note that revelation of both w and k is not an issue, because they alone do not suffice for authentication, and their role is simply to vouch for the validity of the credential in use. In the **Authentication Protocol**, the user needs to extend $\Pi_{\textit{R-BBS}}$ in the basic scheme to include $\Pi_{acc} = PoK\{(w,k) : e(w, W_{acc} \cdot \hbar^k) = e(\Lambda, \hbar)\}$. We denote the resulting proof as Π_1, where $\Pi_1 = PoK\{(M, k, s, u, w) : e(M, W \cdot h^k) = e(a,h)^u \cdot e(b,h)^s \cdot e(d,h) \bigwedge e(w, W_{acc} \cdot \hbar^k) = e(\Lambda, \hbar)\}$. We omit the details, as it is a direct combination of $\Pi_{\textit{R-BBS}}$ and Π_{acc}.

In case of user dynamics, e.g., an existing user withdraws or a new user enrols in, each valid user can generate a new witness by herself, with the help of her old witness, her own k, together with the new accumulator Λ and the event trigger k (that causes the update) published by the server. The details can be found in [2, 24].

5. COUNTERMEASURES TO ONLINE GUESSING ATTACKS

Recall that online guessing attack is inherent, impossible to be eliminated at the protocol level, due to the low entropy of passwords. However, in regular password authentication, system-level measures, e.g., freezing a victim user's account, are effective. In contrast, online guessing attack becomes particularly troublesome in anonymous password authentication (both the password-protected credential approach and the conventional approach), where discrimination of users is not allowed. In this section, we propose a *virtual* TTP (vTTP) based strategy to counter against online guessing attacks.

5.1 Intuitions

To lead to the final vTTP-based method, we begin with a discussion of several potential methods that may be useful in mitigating online guessing attacks in certain applications. This also serves to help better understand the challenges posed by online guessing attacks in the setting of anonymous password authentication.

All-Or-Nothing Method. Without being able to discern individual users, a straightforward method is to freeze all user accounts in case of online guessing attacks. In particular, the server counts failed login attempts, and locks all users' accounts as long as the total number of failures thus far suffices to endanger a user's password (assuming all failed logins are targeted at the same user). However, this all-or-nothing method is clearly not satisfactory, as it affects all users.

Enforcement of Short-term Credentials. An alternative method is to alleviate the consequences of online guessing attacks even when they succeeded. The idea is to enforce short-term credentials, such that even if an attacker recovers a user's password and credential by online guessing attacks, he cannot use the credential for long. Specifically, we mandate that a user credential is valid only for a limited period of time, and it expires after the period. Technical details follow. A credential has a expiry time τ, so the credential for user u is a BBS signature (M, k, s), where $M = (a_0^u \cdot a_1^\tau \cdot b^s \cdot d)^{\frac{1}{k+\chi}}$, and $a_0, a_1 \in G_1$ are public parameters of the BBS signature. Consequently, τ is also left un-protected in the password-protected credential, which is then $\langle u, k, w, \tau, \widehat{cred}_1 = [M]_{pw}, \widehat{cred}_2 = \texttt{E}(s)\rangle$. At the time of login, Π_1 is extended to be $PoK\{(M, k, s, u, w, \tau) : e(M, W \cdot$

$h^k) = e(a_0, h)^u \cdot e(a_1, h)^\tau \cdot e(b, h)^s \cdot e(d, h) \bigwedge e(w, W_{acc} \cdot \hbar^k) = e(\Lambda, \hbar) \bigwedge \tau \geq T\}$, where T is current time.

However, the main problem with this method is how to determine an appropriate validity period for a credential: if too short, then the user would be forced to frequently update credentials; if too long, the intended goal of damages-control would be constrained.

TTP-based Anonymity Revocation. Following other privacy preserving primitives such as group signature and anonymous credential (e.g., [15, 17]), another alternative is to allow the server to revoke user anonymity in case of attacks, so that the server can inform the victim user or withdraw her credential. Anonymity revocation is normally performed by a TTP in other privacy-preserving primitives. We observe that in some applications of password authentication, a TTP-based structure indeed exists. For example, in a federated enterprise composed of a headquarter and many branches, it is reasonable to expect the headquarter to be the TTP for its subsidiaries.

We thus give a method, assuming the presence of a (offline) TTP for anonymity revocation. Suppose that the TTP holds public key encryption, e.g., ElGamal encryption, where $\mathtt{E}_{TTP}(.)/\mathtt{D}_{TTP}(.)$ denote encryption/decryption, respectively. Then at the time of login, the user additionally sends $E = \mathtt{E}_{TTP}(\xi^u)$ to the server in step 1, where $\xi \in G_1$, and Π_1 is extended to be $PoK\{(M, k, s, u, w) : e(M, W \cdot h^k) = e(a, h)^u \cdot e(b, h)^s \cdot e(d, h) \bigwedge e(w, W_{acc} \cdot \hbar^k) = e(\Lambda, \hbar) \bigwedge E = \mathtt{E}_{TTP}(\xi^u)\}$. Consequently, when online guessing attacks arise, the server asks the TTP to recover ξ^u from E.

Here ξ^u is called *identity tag*, from which the owner u can be located. To facilitate mapping from ξ^u to u, the server maintains a table of pairs of (ξ^u, u) for all users. Note that the reason why we choose to "identity escrow" ξ^u rather than u is simply for ease of zero-knowledge proof.

5.2 Details of vTTP

The above TTP-based method is promising, provided that a TTP presents. However, in the setting of password authentication, there does not always exist a TTP. Without the assistance of TTPs for anonymity revocation, an alternative in the literature is "self-enforcing" anonymity revocation, in the sense that the server itself can revoke user anonymity, in case unusual events occur. We have checked some of the primitives of this nature in the literature, e.g., [2, 13, 33], among many others, and found that their techniques cannot be applied in our case. The reason is that in their systems, to enable self-enforceable anonymity revocation, some elements of a user credential must be committed to *deterministically* in the zero-knowledge proof of credential showing. However, in our case since the server knows all credentials, every element of a credential must be committed to in a *randomized* form (e.g., T_1, T_2 in $\Pi_{\text{R-BBS}}$); otherwise, the server can enumerate all credentials against the commitment to determine the actual credential in question. Although we do not know for sure whether deterministic commitment is a necessary condition to achieve self-enforcing anonymity revocation, it seems the case!

Virtual TTP. Our final strategy is *virtual* TTP (vTTP) based anonymity revocation, enlisting the help of users for anonymity revocation. In particular, the server and users form a *virtual* TTP, replacing the real TTP in the above TTP-based method. The basic idea is to secret-share the private key of TTP among the server and all N users, such that the server together with a number of users (e.g., t-out-of-N) can open identity escrow. Indeed, users have motive in participating in anonymity revocation, as any one of them can be the victim. For this method to work, the following properties must be satisfied:

P1. The server alone cannot revoke user anonymity.

P2. Even if all users collude, they cannot revoke user anonymity.

Key Generation. To fulfil the above properties, we need a public key encryption: (1) there is no trusted "dealer" that helps distribute shares of the private key to the shareholders; (2) the private key should not be reconstructed on any entity at any time. The *dynamic threshold cryptosystems*, e.g., [18, 19, 25, 29], have these features. The ElGamal-type schemes in [29, 25] fit our context. The major issue to be addressed is how to generate the key pair for the vTTP, to meet **P1** and **P2**. Suppose each user has a key pair $(pk_i, sk_i) = (g^{x_i} \in G_1, x_i \in Z_q)$ for ElGamal encryption[2]; also, suppose the server re-uses his ElGamal encryption, and let the key pair be $(PK_{\text{Server}} = g^{x_S}, SK_{\text{Server}} = x_S)$. The idea of generating the key pair $(PK_{v\text{TTP}}, SK_{v\text{TTP}})$ for the vTTP is depicted in Figure 1(a): first, the N users (who are willing to participate in anonymity revocation) coordinate to generate the public key $PK_{\text{U}} = g^{x_U}$ using the dynamic threshold encryption scheme (e.g.,[25]), such that t-out-of-N users can decrypt ciphertexts generated under PK_{U}, while without construction of $SK_{\text{U}} = x_U$[3]. Then the server computes $PK_{v\text{TTP}} = PK_{\text{Sever}} \cdot PK_{\text{U}} = g^{x_S + x_U}$. The corresponding $SK_{v\text{TTP}}$ should be $x_S + x_U$, but it is never really computed. With the presence of $PK_{v\text{TTP}}$, the authentication protocol between a user and the server remains the same as in the above TTP-based method, with the only exception of vTTP replacing TTP.

In principle, dynamic threshold cryptosystems allow for user withdrawal/enroment, but that involves all users to update their key shares. To mitigate this issue, an alternative key generation strategy is depicted in Figure 1(b): N users are partitioned into t groups, and each group generates a key pair $(PK_i, SK_i), i = 1, 2, \cdots, t$, under the dynamic threshold encryption scheme. Then $PK_{v\text{TTP}} = PK_{\text{Server}} \cdot PK_1 \cdots PK_t$, and $SK_{v\text{TTP}} = SK_{\text{Server}} + SK_1 + \cdots + SK_t$. Anonymity revocation requires a cooperation among the server and the t groups. In this method, update of key shares due to user dynamics is restricted to a particular group, without affecting other groups. Again, do not confuse user dynamics in this context with that in password authentication, and they are not necessarily relevant: a user can be permitted to remain on board for anonymity revocation as long as she are still happy to do so, even she has withdrawn from the password authentication system; and vice versa (see also footnote 2).

Abuse of Anonymity Revocation. An issue remains to be discussed is how to prevent the server from abusing this

[2] We remind not to confuse this key held by a user with her credential. This key is independent of password authentication, and it should be managed in a secure facility. This, however, does not negate the portability of password-protected credential, as the key is not used for authentication. Indeed, if a user is not willing to help in anonymity revocation, then she does not need to have this key.

[3] In fact, to decrypt a ciphtertext, each user generates a *partial* ciphertext by applying her own private key, and t or more partial ciphertexts appropriately combined together yields the plaintext.

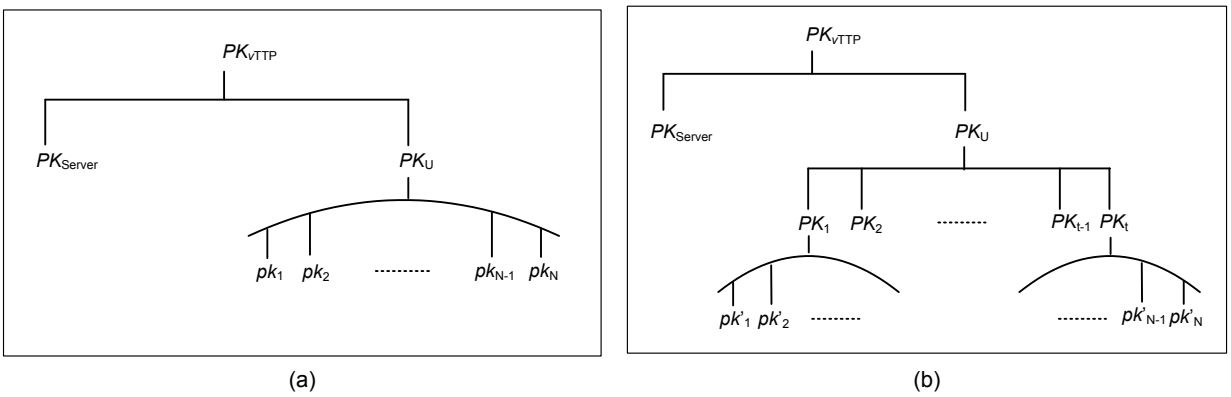


Figure 1: Key Generation Strategies

anonymity revocation mechanism, e.g., the server asks to open the identity escrow of a valid login. A simple solution is mandating the server to publish all revealed user identities, together with the corresponding protocol transcripts, in a public directory where all users are free to access. As such, abuses by the server can be found out by users. This will become clearer shortly in our proof-of-concept implementation (see Section 7).

Discussions (Weakness of vTTP). We remark that a weakness of this vTTP-based anonymity revocation is that it requires *timely* assistance from users. Thus in practice, the promptness of anonymity revocation cannot always be guaranteed. We leave it as an open problem to find more effective countermeasures than the vTTP-based method.

6. SECURITY DEFINITION AND ANALYSIS

In this section, we formulate the security notions for password-protected credentials, and then analyze the security of our scheme.

Adversary Model. Two types of adversaries are considered: outsiders and the server, with respect to different security objectives defined below. An outsider adversary is defined to be a coalition of any entities (including especially other valid users), other than the server and the user who is under attack.

Security Definition. The following security notions are defined for password-protected credential systems.

Security of Password. The adversary that is interested in acquiring user passwords is outsiders. Indeed, it is not possible to prevent the server from learning user passwords by offline guessing attacks, given the verifiability of credentials to the server. However, exposure of user passwords to the server is not an issue in the context of password authentication. We distinguish between *passive* and *active/malicious* outsider adversary. The foremost security requirement for password-protected credentials is that by offline guessing attacks, a passive outsider adversary learns nothing on the underlying password from a password-protected credential, as well as by eavesdropping on the authentication protocol between the (credential) owner and the server. This notion is captured by a simulator $\mathcal{S}$ which can interact with the adversary, in disguise of the owner, although it has no knowledge of her password-protected credential and password. Formally, we play a game between a challenger and a PPT (Probabilistic Polynomial Time) adversary. The challenger $\mathcal{C}$ sets up the system by invoking Setup, and enrols users by executing Registration. Afterwards, $\mathcal{C}$ gives the public system parameters and all but u_j's password-protected credentials to the adversary $\mathcal{A}$, who is also offered the following oracles:

`GetCred`(i): $\mathcal{C}$ returns to $\mathcal{A}$ the credential of user u_i, for any $i \neq j$.

`ExeAuth`(j): $\mathcal{C}$ tosses a coin to decide whether to use u_j or $\mathcal{S}$. If the former, $\mathcal{C}$ simply invokes Authentication Protocol between u_j and the server, and sends the protocol script together with u_j's password-protected credential to $\mathcal{A}$. Otherwise, $\mathcal{C}$ returns to $\mathcal{A}$ the simulated transcript by $\mathcal{S}$, which includes the simulated authentication protocol transcript and a simulated password-protected credential. We say that a system achieves *security of password with respect to passive outsider adversary*, if $\mathcal{A}$ cannot distinguish whether $\mathcal{C}$ is using u_j or $\mathcal{S}$.

For an active outsider adversary, we know that an unavoidable attack is that the adversary $\mathcal{A}$ first obtains a "credential" by opening the password-protect credential with a guessed password; then engages in Authentication Protocol with the server using the "credential", and validates the guess by seeing the accept/reject feedback from the server. We say that a system achieves *security of password with respect to active outsider adversary*, if the best $\mathcal{A}$ can achieve in validating its guesses of password is by taking advantage of such online guessing attacks.

Unlinkability. The adversary against unlinkability is the server. It requires that the server cannot link different logins by the same user. This notion is, again, captured by a game between a challenger and a PPT adversary as below. The challenger $\mathcal{C}$ sets up the system and enrols users by executing Setup and Registration, respectively, and gives all generated information, be it public or private, including all user credentials to the server adversary $\mathcal{A}_S$ (except the private key of vTTP). Afterwards, for a challenge user u_j, $\mathcal{C}$ uses either u_j or a simulator $\mathcal{S}$, depending on a coin toss, to engage in Authentication Protocol with $\mathcal{A}_S$. We say that a system achieves *unlinkability*, if $\mathcal{A}_S$ cannot distinguish whether $\mathcal{C}$ is using u_j or $\mathcal{S}$.

Authenticated Key Exchange. The adversary against the Authentication Protocol is clearly outsiders. An outsider adversary can try to attack *entity authentication* or *key secrecy*

of the protocol, other than exposing user password. In the literature, formulation of such authenticated key exchange protocols is well studied, e.g., [6, 9, 12, 14], to name but a few. It is not hard to adapt these existing models such as [6] to our case, and the details are omitted due to the limited space. However, an important note is that session corruption is not allowed in our model: if, by corrupting a session, the adversary learns the random coins used in the commitments of credential elements, then the adversary can de-randomize the commitments and in turn perform offline guessing attacks against the de-randomized quantities.

Security Analysis. We analyze the security of our final scheme (i.e., basic scheme + membership withdrawal + vTTP-based anonymity revocation), and have the following theorem.

THEOREM 1. *Our scheme achieves security of password, unlinkability, and authenticated key exchange.*

PROOF. (Sketch) (1) *Security of Password*: For passive adversary, it suffices to show how to construct the simulator. Let $\Pi_2 = PoK\{(M,k,s,u,w) : e(M, W \cdot h^k) = e(a,h)^u \cdot e(b,h)^s \cdot e(d,h) \bigwedge e(w, W_{acc} \cdot \hbar^k) = e(\Lambda, \hbar) \bigwedge E = \mathtt{E}_{vTTP}(\xi^u)\}$, which is the zero-knowledge proof by the user in our scheme. The real transcript (between a user and the server) that the challenger gives to the adversary is $[s^*, X, N_A^*, \mathtt{E}_{vTTP}(\xi^u), \mathtt{Cmt}(\Pi_2), N_B, Y, \mathtt{Mac}, \mathtt{Res}(\Pi_2), \langle u, k, w, [M]_{pw}, \mathtt{E}(s)\rangle]$. To imitate the transcript, the simulator selects $\bar{s}^* \in_R G_1^2, \bar{X} \in_R G_1, \bar{N}_A^* \in_R G_1^2, \bar{E} \in_R G_1^2, \bar{Y} \in_R G_1, \overline{\mathtt{Mac}} \in_R \{0,1\}^{\kappa_1}, \overline{M} \in_R [.], \overline{s}_E \in_R G_1^2$, and invokes the simulator of Π_2 to generate $\overline{\mathtt{Cmt}}(\Pi_2), \bar{c}, \overline{\mathtt{Res}}(\Pi_2)$. In addition, the simulator also generates a pair $(\bar{w}, \bar{k})$ by invoking the extractor of Π_{acc}. Then the output of the simulator is $[\bar{s}^*, \bar{X}, \bar{N}_A^*, \bar{E}, \overline{\mathtt{Cmt}}(\Pi_2), \bar{c}, \bar{Y}, \overline{\mathtt{Mac}}, \overline{\mathtt{Res}}(\Pi_2), \langle u, \bar{k}, \bar{w}, \overline{M}, \overline{s}_E\rangle]$. We show that the simulator's output is computationally indistinguishable from the real transcript. Due to the semantic security of ElGamal encryption, $\bar{s}^*, \bar{N}_A^*, \bar{E}, \bar{s}_E$ is computationally indistinguishable from $s^*, N_A^*, \mathtt{E}_{vTTP}(\xi^u), \mathtt{E}(s)$; similarly, other elements can also be shown indistinguishable under appropriate (computational) assumptions. Of special attention is $\overline{M}$ and $[M]_{pw}$: we must guarantee that M is uniformly random over G_1, and the distribution of $[M]_{pw}$ is uniform over the range of [.] for all possible pw. Indeed, the former is the case due to the BBS signature, and for the latter, any practical symmetric key encryption scheme has that property.

For active adversary, we argue that in our protocol the only potential leakage of the credential information is through s^* and Π_2. As such, active attacks are no better than passive eavesdropping, and thus the only way left for the adversary to validate his guesses of the target password is by interacting online with the server.

(2) *Unlinkability*: Given the above proof, how to construct the simulator is straightforward, and we omit the details. We just want to stress that the server adversary is much more powerful than the above outsider adversary, as it has been given virtually all system information (public and private), including especially the challenge user's credential. The insight why such a powerful adversary still cannot distinguish is that every element of the user credential is committed to in a randomized form, e.g., T_1, T_2 in $\Pi_{\hbar\text{-}BBS}$, which makes a PPT adversary unable to distinguish without the knowledge of the random coins.

(3) *Authenticated Key Exchange*: We can prove this property in a weaker variant of the model [6] by not allowing for session corruption. Our Authentication Protocol essentially is a combination of the signature based authenticator and the (public key) encryption based authenticator in [6]. Security of our protocol (in terms of authenticated key exchange) thus follows from the security of the two authenticators, according to the rationale of the modular protocol design and analysis [6][4]. It should be noted that the encryption based authenticator is proven secure in [6], given that the encryption is CCA2 secure; while in our protocol, the ElGamal encryption (i.e., $\mathtt{E}(.)$) is only CPA secure. This is not an issue, as session corruption is not allowed in our model and if both entities immediately erase their local states (e.g., N_A), once the shared key is established. □

7. IMPLEMENTATION RESULTS

We have two objectives for our implementation: (1) empirically evaluate the efficiency of our scheme; (2) provide a proof-of-concept prototype, illustrating the usage of our proposal in practice.

7.1 Performance Evaluation

We conducted extensive experiments on the efficiency of our (final) scheme. The programs were written in C/C++, and the underlying elliptic curves and pairing operations were implemented based on the Miracl library[5]. In particular, we used DDH-hard subgroups of an MNT elliptic curve with pairings, where $|q| = 159$ bits, and the curve has an embedding degree of 6. Our implementation also made use of HMAC, SHA-1, PRNG (Pseudorandom Number Generator) in the Miracl library, and our own codes of ElGamal-type encryption. For the experiments, the client program runs on a Fujitsu notebook, Intel Core2 Duo CPU, 2.53GHz, OS Windows XP, and the server program runs on a PC, Dell Dimension 9150, Intel 3.0 GHz CPU, 1GB RAM, OS Windows XP. We experimented with different number of users simultaneously authenticating with the server, and the average time it takes for a user, respectively, and the server to complete the authentication protocol is reported below.

	User	**Server**
Timing (ms)	385	430

The results indicated that our scheme has high efficiency, should be acceptable for practical applications.

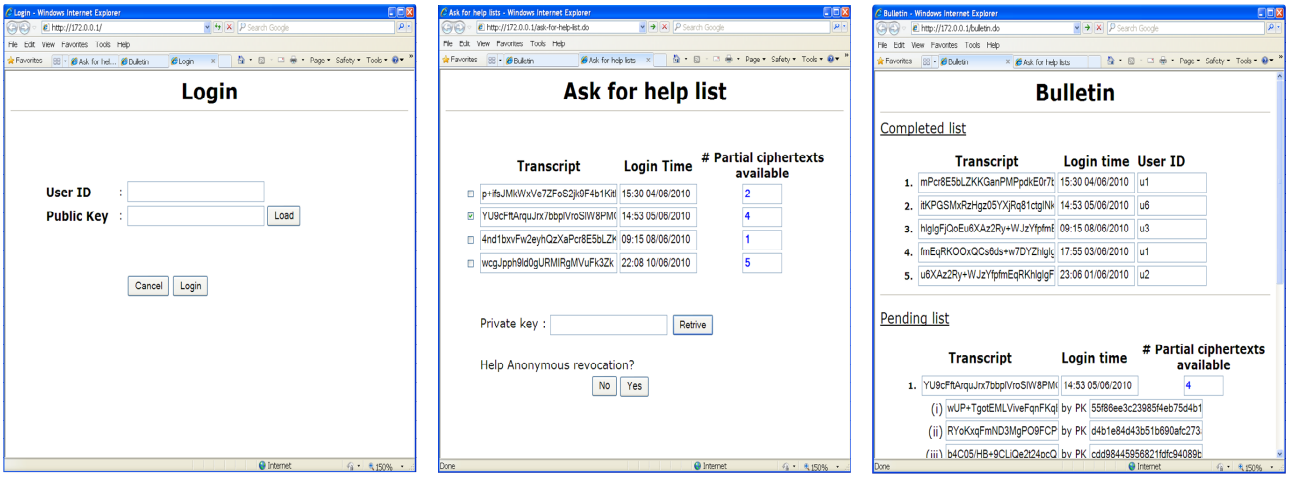
7.2 Proof-of-Concept Prototype

For the prototype, we mainly provide Web-based interfaces to implement the vTTP-based anonymity revocation. It is our hope that the prototype can serve as a simple demon to illustrate how our proposal can be used in practice.

Login interface. To participate in anonymity revocation, a user first logs in to the system through the login interface shown in Figure 2(a), where the user needs to provide her identity, her share of the vTTP public key, together with a short (zero-knowledge) proof of the corresponding private key. At the server side, the server keeps a repository of all public keys whose owners have agreed to offer assistance for anonymity revocation.

[4]We noticed that some (minor) attacks have been found against the two authenticators, but incidentally, all are due to session corruption.

[5]M.Scott. Indigo software: http://indigo.ie/∼mscott.



(a) (b) (c)

Figure 2: User Interfaces

Ask-for-Help list. Once her login request is verified, the user is navigated to the Ask-for-Help interface, shown in Figure 2(b), which lists all the protocol transcripts that need the user's help for anonymity revocation. Notice that each entry is succeeded by the number of partial ciphertexts that have already been collected; also, the user is offered the option to select which entries she is interested in. The user then retrieves her private key from the storage and clicks the "Yes" button to start decryption of the selected protocol transcripts one by one. Each resulting partial ciphertext will be sent to the server, who then updates the Bulletin (see below). The user can immediately check the bulletin to see whether her results are correctly reflected in the bulletin.

Public bulletin. The bulletin is free of access to the public, and it consists of a *Completed List* and a *Pending List*, as shown in Figure 2(c). The completed list contains the set of protocol transcripts, wherein the involved user identities have been successfully revealed. The pending list shows the protocol transcripts that have not collected sufficient number of partial ciphertexts from users. For each pending item, all of its collected partial ciphertexts are displayed below it. This facilitates users who have contributed to anonymity revocation to check whether their partial ciphertexts are correctly handled by the server. Once there are sufficient number of partial ciphertexts, the server performs the final decryption and moves the item to the completed list. The pending list serves as an important deterrence to discourage the server from abuses of anonymity revocation.

8. CONCLUSION AND FUTURE WORK

We addressed two issues (membership withdrawal and online guessing attacks) which have not been considered in Yang *et al.*'s anonymous password authentication scheme, thus advancing the primitive of anonymous password authentication a step further towards practicality. We adopted a set of different building primitives in our scheme, and achieved much better efficiency (both analytically and empirically). We proved the security of our scheme, and implemented a proof-of-concept prototype.

Recall that as in [36], we still relied on the server side's homomorphic encryption to achieve limited verifiability of user credentials. An alternative we have in mind (which can avoid the use of homomorphic encryption) is that the server does not publish the public key of the BBS signature, so that the signatures cannot be publicly verified. We have checked several options (e.g., keeping secret W, or a, b, d, or h), and some of them seem work. As a future direction, we will continue to explore along this line.

9. ACKNOWLEDGMENTS

This work is supported by the A*STAR project SEDS-0721330047. We thank Huang Xinyi for his comments on the use of ElGamal homomorphic encryption in our case.

10. REFERENCES

[1] M. Abdalla, M. Izabachene, and D. Pointcheval. Anonymous and transparent gateway-based password-authenticated key exchange. In *Proc. International Conference on Cryptology and Network Security, CANS'08*, pages 133-148, 2008.

[2] M.H. Au, W. Susilo, and Y. Mu. Constant-size dynamic k-TAA. In *Proc. Security and Cryptography for Networks, SCN'06*, LNCS 4116, pages 111-125, 2006.

[3] F. Boudot. Efficient proofs that a committd number lies in an interval. In *Proc. Advances in Cryptology, Eurocrypt'00*, LNCS 1807, pages 431-444, 2000.

[4] X. Boyen. Hidden credential retrieval from a resuable password. In *Proc. ACM. Symposium on Information, Computer and Communications Security, ASIACCS'09*, pages 228-238, 2009.

[5] D. Boneh, X. Boyen, and H. Shacham. Short group signatures. In *Proc. Advances in Cryptology, Crypto'04*, LNCS 3152, pages 41-55, 2004.

[6] M. Bellare, R. Canetti, and H. Krawczyk. A modular approach to design and analysis of authentication and key exchange protocols. In *Proc. ACM. Annual Symp. on Theory of Computing, ToC'98*, pages 419-428, 1998.

[7] E. Bresson, O. Chevassut, and D. Pointcheval. Security proofs for an efficient password-based key

exchange. In*Proc. ACM. Computer and Communication Security, CCS'03*, pages 241-250, 2003.

[8] S. Bellovin, and M. Merritt. Encrypted key exchange: password-based protocols secure against dictionary attacks. In *Proc. IEEE Symposium on Research in Security and Privacy, S&P'92*, pages 72-84, 1992.

[9] S. Blake-Wilson, and A. Menetzes. Entity authentication and authenticated key transport protocol employing asymmetric techniques. In *Proc. Security Protocols Workshop*, LNCS 1361, pages 137-158, 1997.

[10] V. Boyko, P. Mackenzie, and S. Patel. Provably secure password-authenticated key exchange using Diffie-Hellman. In *Proc. Advances in Cryptology, Eurocrypt'00*, LNCS 1807, pages 156-171, 2000.

[11] M. Bellare, D. Pointcheval, and P. Rogaway. Authenticated key exchange secure against dictionary attacks. In *Proc. Advances in Cryptology, Eurocrypt'00*, pages 139-155, 2000.

[12] M. Bellare, and P. Rogaway. Entity authentication and key distribution. In *Proc. Advances in Cryptology, Crypto'93*, LNCS 773, pages 232-249, 1993.

[13] J. Camenisch, S. Hohenberger, and A. Lysyanskaya. Compact e-cash. In *Proc. Advances in Cryptology, Eurocrypt'05*, LNCS 3494, pages 302-321, 2005.

[14] R. Canetti, and H. Krawczyk. Analysis of key exchange protocols and their use for building secure channels. In *Proc. Advances in Cryptology, Eurocrypt'01*, LNCS 2045, pages 451-472, 2001.

[15] J. Camenisch, and A. Lysyanskaya. Efficient non-transferable anonymous multi-show credential system with optional anonymity revocation. In *Proc. Advances in Cryptology, Eurocrypt'01*, LNCS 2045, pages 93-118, 2001.

[16] J. Camenisch, and A. Lysyanskaya. A signature scheme with efficient protocols. In *Proc. Security and Cryptography for Networks, SCN'02*, LNCS 2576, pages 268-289, 2002.

[17] J. Camenisch, and M. Stadler. Efficient group signature schemes for large groups. In *Proc. Advances in Cryptology, Crypto'97*, LNCS 1296, pages 410-424, 1997.

[18] C. Delerabl, and D. Pointcheval. Dynamic threshold public-key encryption. In *Proc. Advances in Cryptology, Crypto'08*, LNCS 5157, pages 317-334, 2008.

[19] R. Gennaro, S. Halevi, H. Krawcyzk, and T, Rabin. Threshold RSA for dynamic and ad-hoc groups. In *Proc. Advances in Cryptology, Eurocrypt'08*, LNCS 4965, pages 88-107, 2008.

[20] S.D. Galbraith, K.G. Paterson, and N.P. Smart. Pairings for cryptographers. Cryptology ePrint Archive: http://eprint.iacr.org/2006/165, 2006.

[21] S. Halevi, and H. Krawczyk. Public-key cryptography and password protocols. In *Proc. ACM. Computer and Communication Security, CCS'98*, pages 122-131, 1998.

[22] D. Hoover, and B. Kausik. Software smart cards via cryptographic camouflage. In *Proc. IEEE Symposium on Security and Privacy, S&P'99*, pages 02-08, 1999.

[23] J. Katz, R. Ostrovsky, and M. Yung. Efficient password-authenticated key exchange using human-memorable passwords. In *Proc. Advances in Cryptology, Eurocrypt'01*, LNCS 2045, pages 475-494, 2001.

[24] L. Nguyen. Accumulators from bilinear pairings and applications. In *Proc. CT-RSA'05*, LNCS 3376, pages 275-292, 2005.

[25] N. Noack, and S. Spit. Dynamic threshold cryptosystem without group manager. Network Protocols and Alorithms, 1(1): 108-121, 2009.

[26] M. H. Nguyen, and S. P. Vadhan. Simpler session-key generation from short random passwords. In *Proc. Theory of Cryptography, TCC'04*, pages 428-445, 2004.

[27] R. Perlman, and C. Kaufman. Secure password-based protocols for downloading a private key. In *Proc. Network and Distributed Systems Security Symposium, NDSS'99*, 1999.

[28] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proc. Advances in Cryptology, Eurocrypt'99*, pages 223-238, 1999.

[29] T.P. Pedersen. A threshold cryptosystem without a trusted party. In *Proc. Advances in Cryptology, Eurocrypt'91*, pages 522-526, 1991.

[30] R. Sandhu, M. Bellar, and R. Ganesan. Password enabled PKI: virtual smartcards vs. virtual soft tokens. In *Proc.* 1^{st} *Annual PKI Research Workshop*, pages 89-96, 2002.

[31] S. Shin, K. Kobara, and H. Imai. A secure construction for threshold anonymous password-authenticated key exchange. IEICE Transactions on Fundamentals, E91-A(11): 3312-3323, 2008.

[32] J. Tardo, and K. Alagappan. SPX: global authenticaiton using public key certificate. In *Proc. IEEE Sysmposium on Security and Privacy, S&P'91*, pages 232-244, 1991.

[33] P. Tsang, M. Au, A. Kapadia, and S.W. Smith. Blacklistable anonymous credentials: blocking misbehaving users without TTPs. In *Proc. ACM. Computer and communications security, CCS'07*, pages 72-81, 2007.

[34] D. Q. Viet, A. Yamamura, and T. Hidema. Anonymous password-based authenticated key exchange. In *Proc. Advances in Cryptology, Indocrypt'05*, LNCS 3797, pages 233-257, 2005.

[35] J. Yang, and Z. Zhang. A new anonymous password-based authenticated key exchange protocol. In *Proc. Advances in Cryptology, Indocrypt'08*, LNCS 5365, pages 200-212, 2008.

[36] Y.J. Yang, J.Y. Zhou, J. Weng, and F. Bao. A new approach for anonymous password authentication. In *Proc. 25th Annual Computer Security Applications Conference, ACSAC'09*, pages 199-208, 2009.

Securing Interactive Sessions Using Mobile Device through Visual Channel and Visual Inspection

Chengfang Fang
School of Computing
National University of Singapore
Singapore
c.fang@comp.nus.edu.sg

Ee-Chien Chang
School of Computing
National University of Singapore
Singapore
changec@comp.nus.edu.sg

ABSTRACT

Communication channel established from a display to a device's camera is known as *visual channel*, and is helpful in securing key exchange protocol [16]. In this paper, we study how visual channel can be exploited by a network terminal and mobile device to jointly verify information in an interactive session, and how such information can be jointly presented in a user-friendly manner, taking into account that the mobile device can only capture and display a small region. Motivated by applications in Kiosk computing and multi-factor authentication, we consider three security models: (1) the mobile device is trusted, (2) at most one of the terminal or the mobile device is dishonest, and (3) both the terminal and device are dishonest but they do not collude or communicate. We give a few protocols and investigate them under the abovementioned models. We point out a form of replay attack that renders some other straightforward implementations cumbersome to use. To enhance user-friendliness, we propose a solution using *visual cues* embedded into the 2D barcodes and incorporate the framework of "augmented reality" for easy verifications through visual inspection. We give a proof-of-concept implementation to show that our scheme is feasible in practice.

Keywords

Visual channel, Sub-region authentication, 2D-barcodes, User-friendly verification.

1. INTRODUCTION

Securing connection to a server through an untrusted network terminal is challenging even if the user has additional factor for authentication like one-time-password token, smartcard, or a mobile phone. One of the hurdles is the difficulty in securely passing information from the terminal to the device, and presenting the jointly verified authentic information to the user in a user friendly manner. Using traditional channel to connect the device and the terminal, like wireless connection or plug-and-play connection, are subjected to various man-in-the-middle attacks. Even if a secure channel can be established, it is still not clear how the additional device can help in authenticating subsequent messages rendered on the untrusted terminal's display.

A number of recent works utilize cameras in the mobile devices to provide an alternative realtime communication channel from a display unit to a mobile device: messages are rendered on the display unit in a form of, say 2D barcodes, which are then captured and decoded by the mobile device via its camera. Although such visual channel could be eavesdropped by "over-the-shoulder" attacks, it is arguably impossible to modify or insert messages, and thus secure against man-in-the-middle attack. Visual channel has been exploited in a few works in verifying the session key exchanged over an unsecured channel, for instance seeing-is-believing proposed by McCune et al. [16]. There are also proposals on verifying untrusted display, for example, Clarke et al. propose verifying the display screen using stabilized camera device [5]. In this paper, we take a step further by investigating authentication of interactive sessions, with consideration that most cameras are unable to cover the whole screen in a single view with sufficient precision. An example of interactive session is online banking application where a user can browse and selectively view pervious transactions, and carry out new transactions. A typical screenshot would contain important information like the user's account information, and less sensitive information like advertisements, help information, and navigation information, as shown in Figure 1(a).

During an interaction session, after a session key k_s has been securely established between the server and the mobile device (could be established using seeing-is-believing [16]), there could be many subsequent communication messages that require protection by k_s. These messages may need to be rendered over different pages, or in a scrolling webpage where not all of them are visible at the same time. We remark that it is not clear how to protect them. For instance, one may render the messages as 2D barcodes, each protected by the same k_s. To view the message in a 2D barcode, the user moves the mobile device over the barcode, and the device will capture, authenticate and display the message on its display panel. However, as there are many barcodes associated with the same key, it is possible for a dishonest terminal to perform "rearrangement" attack: replays barcodes or shows barcodes in the wrong order.

The above attack arises due to the limitation that the camera is unable to capture the whole screen with sufficient precision. We treat the problem as the authentication of

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

messages rendered in a sequence of large 2D regions, where only region in a small rectangular window can be captured at one time. There are a few straightforward methods to overcome the rearrangement attack. For instance, one may prevent the attack by requiring the user to scan all the barcodes with his mobile device, and all the messages will be authenticated and rendered by the mobile device. However, it is troublesome for the user to scan all the barcodes, and there are situations where the user only wants to view some, but not all, of the messages. In addition, it is less preferred to navigate and browse the messages (e.g. a large table of transactions) within the relatively small display panel. In Section 6, we will discuss a few other straightforward methods and their limitations.

Our solution is to use a barcode scheme that given a message m and a *visual cue* v, is able to produce a barcode image that not only carries m as its payload, but also visually appears as v (see examples in Figure 1(b) and Figure 1(c)). Our paper realizes such barcode scheme using technique borrowed from fragile image watermarking [15], where the visual cue is the "host" image, and the payload is embedded as the "watermark". To embed a long message into several barcodes, our main idea is to have a visual cue on each barcode indicating its position. By visually inspecting the visual cues, the user can readily verify that the barcodes are in the correct arrangement. For example, in Figure 1(b), the visual cues are numeric numbers increasing by 1 from left to right, top to bottom. The black dot beside the number "2" indicates that the barcode is at the end of the row, and the black block beside the number "8" indicates that it is the last (i.e. bottom-right) barcode. With the arrangement of barcodes verified, the user can then browse selective barcodes independently with his mobile device.

In our security analysis, we consider the four parties setting where a user, who has a mobile device, wants to interact with a server via a network terminal. We focus on three security models. In the first model, the network terminal, including its CPU, keyboard and display unit, is untrusted by the user, whereas the mobile device is trusted. This model is motivated by the challenging problem in securing Kiosks [11, 13], where Kiosks are untrusted public network terminal like workstations in Internet café.

In the second model, motivated by two-factor authentication, we consider scenarios where both the mobile and the terminal are not trusted by the user, but at least one of the terminal or mobile carries out the protocol honestly. This is to reflect the concern that either the terminal or the mobile could be, but less likely both are, compromised. We found that under the first model, it is possible to provide both confidentiality and authenticity; whereas under the second model, although authenticity can be achieved, it is not clear how to achieve confidentiality.

In the third model, we take one step further and consider a tricky setting where both the terminal and mobile device could be dishonest, but they do not collude in the sense that they do not know how to communicate with each other. This model is motivated by scenarios where the terminal and mobile device are compromised, but independently by two different adversaries, for instance, a dishonest mobile device that always says "authentic" for whatever authentication it is supposed to carry out, and a network terminal which is remotely controlled by a malicious party who wants to deceive the user to accept a particular message. To detect such dishonest mobile device, our proposed method requires the mobile device to extract and produce a human readable proof from the authentication tag. A corresponding proof is also shown in the terminal's display and hence the user can visually verify whether they are consistent, as shown in Figure 1(c).

In addition to security requirements, user experience is also important. Requiring the user to take snapshot of the screen is rather disruptive from the user's point of view. We employ *augmented reality* to provide better user experience in verification. The design of our 2D barcode and the subregion authentication takes usability into consideration and fits nicely in the framework of augmented reality. One example is as shown in Figure 1(b). The screenshot displayed by the terminal is a combination of sensitive data and non-sensitive data like advertisement and menu. The sensitive data are replaced by 2D barcodes with visual cue as described before. The user treats the mobile device as an inspection device and places the mobile phone over the region to be inspected. In realtime, the mobile device captures and verifies the 2D barcode. If it is authentic, the decrypted message is displayed. The non-sensitive portion of the screenshot is also displayed as it is to help the user to navigate. We give a proof-of-concept system implemented on Android mobile phones and evaluate its performance to show the feasibility of our methods.

Organization

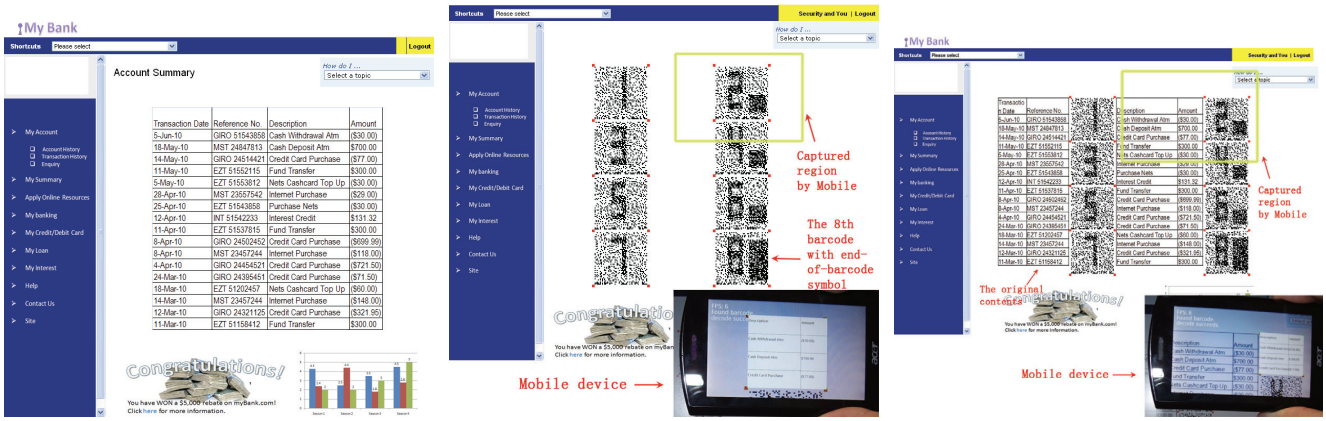
We formally define our problem and three adversary models in Section 2. Assuming the existence of a barcode scheme that is secure against rearrangement attack, we propose two protocols and analyze them under the three adversary models in Section 3. We give a construction for the required barcode scheme using visual cues in Section 4 and discuss the design of visual cue symbols in Section 5. We compare our solutions with possible alternative methods in Section 6. We describe our proof-of-concept implementation in Section 7 and measure its performance in Section 8. A discussion of existing work is given in Section 9. Section 10 gives a conclusion of our paper.

2. MODELS AND FORMULATION

There are four parties involved in our problem: the user, the server, the mobile device and the network terminal. Let us call them `User`, `Server`, `Mobile`, and `Terminal` respectively. In our framework, the term "user" literately refers to a person, and the mobile device is equipped with a camera, input device, a small display unit and sufficient computing power.

A summary of our notations is given in Table 1 and the communication channels among the four parties are as shown in Figure 2. Note that there is no direct communication link between `Mobile` and `Server`. With 3G mobile network and WiFi connection widely available, one may argue that the model should consider such a link. Nevertheless, there are situations where the connection is not available due to cost or other constrains. In addition, there are also security concerns if the mobile device has Internet connection during the transactions: if `Mobile` can directly communicate with a remote malicious party, it may collude and conduct coordinated attack with `Terminal` and the malicious party.

We consider the following security models for the channel between `Server` and `User`:



(a) A bank transaction webpage. (b) Method 1: mobile device is trusted. (c) Method 2: mobile device could be dishonest.

Figure 1: Illustration of our schemes: Figure 1(a) is the bank transaction screenshot which contains a sensitive transaction table to be protected. Figure 1(b) illustrates method 1 where the sensitive table is replaced by barcodes; and the mobile device captures, verifies and decodes part of the table. Figure 1(c) illustrates method 2 where the sensitive table is displayed with barcodes; the user compares the tables on both the terminal and mobile.

1. Model 1: `Terminal` is not trusted by `User`, but `Mobile` is trusted and we want to protect both confidentiality and authenticity.
2. Model 2: At least one of `Terminal` and `Mobile` is honest and we want to protect authenticity.
3. Model 3: Both `Terminal` and `Mobile` could be dishonest but they do not collude and we want to protect authenticity.

In Model 3, we treat the dishonest `Terminal` and `Mobile` as two different adversaries $A_{\mathtt{T}}$ and $A_{\mathtt{M}}$ with two different goals. $A_{\mathtt{T}}$ is the dishonest terminal and its intension is to trick the user to believe that a given message m' is authentic. The actual value of m' is not determined prior to the connection. We can view it as a randomly chosen message that is passed to the $A_{\mathtt{T}}$. The adversary $A_{\mathtt{M}}$ is the dishonest mobile and has an easier goal: it is free to construct any message and trick the user to wrongly believe that it is authentic. An example of $A_{\mathtt{M}}$ is one who always accepts whatever verification it is tasked to do. To capture the notion that they do not collude, we impose the restriction that $A_{\mathtt{T}}$ and $A_{\mathtt{M}}$ do not know how to communicate with each other, and the forge message m' is randomly chosen. Hence, we exclude the attack where $A_{\mathtt{T}}$ covertly sends the message m' to $A_{\mathtt{M}}$ through the visual channel.

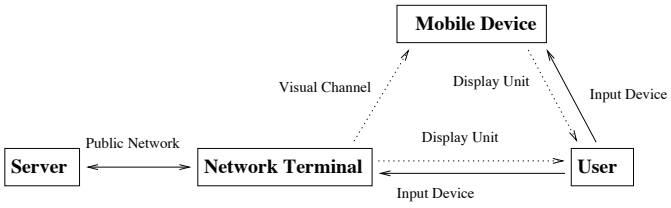


Figure 2: The communication channels among the four parties.

3. PROTOCOLS

We now give our proposed protocols for securing the communication between `Server` and `User` assuming we have a barcode embedding technique that can protect the integrity and confidentiality of its payload, and visible visual cue can be rendered onto the barcode to indicate the barcode location as in Figure 1(b). Given a message m, a visual cue v, and a session key k_s, let us write the barcode (represented as images) as $\mathsf{B}(k_s, m, v)$. For clarity in presentation, we first consider the case where the message can be embedded into one barcode block whose size is small enough to be entirely captured by `Mobile`'s camera with sufficient precision. Thus, we take the visual cue v as a single dot, indicating to the user that there is only a single barcode to be read. We will later study the case for multiple messages in Section 4 and Section 5.

We assume that `Server` has already established a long term shared key with `Mobile` when the user registers an account with the server. In additional, for model 2 and 3, we assume that `User` has established a password with `Server` which is secret to `Mobile`. Before each interactive session, `Server` authenticates `User` and `Mobile` to get a session key k_s, which is to be kept secret from `Terminal`. A secure key exchange can be derived from modified seeing-and-believing [16] and combination of the proposed method in this section. Due to space constrain, we do not include details on key exchange in this paper.

3.1 Server to User

Consider the case where `Server` wants to send a message $m_{\mathtt{S}}$ to `User`. We propose two methods, denoted MS1 and MS2 (message from server), where method MS1 is more user-friendly compared to MS2, but it requires that `Mobile` is trusted.

MS1. To send a message $m_{\mathtt{S}}$ to `User`, the following steps are carried out. (1) `Server` generates a barcode image $\mathsf{B}(k_s,$

Table 1: Summary of Notations.

$m_{\mathtt{U}}$	The message from `User` to `Server`.
$m_{\mathtt{S}}$	The message from `Server` to `User`.
$k_{\mathtt{T}}$	The key for message authentication scheme.
$k_{\mathtt{E}}$	The key for encryption.
$k_{\mathtt{V}}$	The key for embedding visual cue.
k_s	The session key where $k_s = (k_{\mathtt{T}}, k_{\mathtt{E}}, k_{\mathtt{V}})$.
v	A visual cue symbol.
$\mathsf{B}(k_s, m, v)$	A barcode image encoding a message m and visual cue v under key k_s.
$\mathcal{T}_{k_{\mathtt{T}}}(m)$	An authentication tag of a message m under key $k_{\mathtt{T}}$.
$\mathcal{E}_{k_{\mathtt{E}}}(m)$	An encryption of a message m under key $k_{\mathtt{E}}$.
$\mathsf{ECC}(m)$	An error correcting encoding of a message m.
$\mathtt{A} \rightarrow \mathtt{B} : m$	The entity `A` sends a message m to another entity `B`.
$\mathtt{A} \xrightarrow{\mathtt{C}} \mathtt{B} : m$	The entity `A` sends a message m to `B` using `C` as a relay point.

$m_{\mathtt{S}}, v)$ and sends the barcode to `Terminal`. Recall that k_s is the established session key, and v is the appropriate visual cue. (2) `Terminal` displays the barcode. (3) `User` inspects and verifies the visual cue is valid. (4) `Mobile` captures the barcode. (5) If `Mobile` successfully verifies the payload $m_{\mathtt{S}}$ embedded in the barcode, it displays $m_{\mathtt{S}}$. If `Mobile` fails to verify $m_{\mathtt{S}}$, then it displays an error message e.

Below is a summary for MS1:

1. `Server` $\rightarrow$ `Terminal`: $\mathsf{B}(k_s, m_{\mathtt{S}}, v)$;
2. `Terminal` $\rightarrow$ `User`: v;
3. `User` verifies the visual cue v;
4. `Terminal` $\rightarrow$ `Mobile`: $\mathsf{B}(k_s, m_{\mathtt{S}}, v)$;
5. `Mobile` $\rightarrow$ `User`: $m_{\mathtt{S}}$ if $m_{\mathtt{S}}$ is authentic, e otherwise.

MS2. The main difference in this method from the previous MS1 is that, the message $m_{\mathtt{S}}$ is displayed by both `Terminal` and `Mobile`, and thus `User` is able to detect if one of them is dishonest. (1) `Server` first generates a barcode image $\mathsf{B}(k_s, m_{\mathtt{S}}, v)$, then it sends both the barcode image and the message $m_{\mathtt{S}}$ to `Terminal`. (2) `Terminal` displays the barcode, side-by-side with $m_{\mathtt{S}}$. (3) `User` inspects and verifies the visual cue. (4) `Mobile` captures the barcode and rejects if the barcode is not authentic, otherwise, displays $m_{\mathtt{S}}$. (5) `User` reads $m_{\mathtt{S}}$ from `Mobile`'s display panel and `Terminal`'s display. (6) `User` accepts $m_{\mathtt{S}}$ if the $m_{\mathtt{S}}$ in step (2) is consistent with $m_{\mathtt{S}}$ in step (4). Below is a summary for MS2:

1. `Server` $\rightarrow$ `Terminal`: $\mathsf{B}(k_s, m_{\mathtt{S}}, v)$, $m_{\mathtt{S}}$;
2. `Terminal` $\rightarrow$ `User`: v, $m_{\mathtt{S}1}$;
3. `User` verifies v;
4. `Terminal` $\rightarrow$ `Mobile`: $\mathsf{B}(k_s, m_{\mathtt{S}}, v)$;
5. `Mobile` $\rightarrow$ `User`: $m_{\mathtt{S}2}$;
6. `User` accepts $m_{\mathtt{S}1}$ if $m_{\mathtt{S}1} = m_{\mathtt{S}2}$.

3.2 User to Server

Now we consider the following methods MU1 and MU2 (message from user) for sending the message $m_{\mathtt{U}}$ to `Server`. Method MU1 protects both confidentiality and authenticity of the message, whereas method MU2 protects only the authenticity but involves less user operation.

MU1. MU1 consists of the following steps to send a message $m_{\mathtt{U}}$ to `Server`. (1) `User` enters $m_{\mathtt{U}}$ to `Mobile`. (2) `Mobile` computes and shows `User` the encrypted form $\mathcal{E}_{k_{\mathtt{E}}}(m_{\mathtt{U}}) \| \mathcal{T}_{k_{\mathtt{T}}}(\mathcal{E}_{k_{\mathtt{E}}}(m_{\mathtt{U}}))$ in readable characters (for e.g. using `uuencode`). (3) `User` sends displayed string to `Server` through `Terminal`'s input device. (4) `Server` accepts $m_{\mathtt{U}}$ if the tag is valid. Below is a summary for MU1:

1. `User` $\rightarrow$ `Mobile`: $m_{\mathtt{U}}$;
2. `Mobile` $\rightarrow$ `User`: $\mathcal{E}_{k_{\mathtt{E}}}(m_{\mathtt{U}}) \| \mathcal{T}_{k_{\mathtt{T}}}(\mathcal{E}_{k_{\mathtt{E}}}(m_{\mathtt{U}}))$;
3. `User` $\xrightarrow{\mathtt{Terminal}}$ `Server`: $\mathcal{E}_{k_{\mathtt{E}}}(m_{\mathtt{U}}) \| \mathcal{T}_{k_{\mathtt{T}}}(\mathcal{E}_{k_{\mathtt{E}}}(m_{\mathtt{U}}))$;
4. `Server` accepts $m_{\mathtt{U}}$ if the tag $\mathcal{T}_{k_{\mathtt{T}}}(\mathcal{E}_{k_{\mathtt{E}}}(m_{\mathtt{U}}))$ is valid.

MU2. In scenarios where the confidentiality of $m_{\mathtt{U}}$ is not required, we can employ a more user friendly protocol MU2 as follow: (1) `User` enters $m_{\mathtt{U}}$ through `Terminal`'s input device, and `Terminal` forwards $m_{\mathtt{U}}$ to `Server`. (2) `Server` generates a barcode $\mathsf{B}(k_s, m_{\mathtt{U}} \| c, v)$, where c is a randomly generated nonce. `Server` sends the barcode to `Terminal`. (3) `Terminal` displays the barcode, and `User` visually verifies that the visual cue v is correct. (4) `Mobile` captures the barcode and rejects if the barcode is not authentic. (5) `Mobile` renders the message $m_{\mathtt{U}}$ and the nonce c on its display. (6) If $m_{\mathtt{U}}$ is consistent with the message `User` entered in step (1), `User` enters c to `Terminal`, and `Terminal` forwards it to `Server`. (7) `Server` rejects if the nonce c is wrong.

Although involves more steps, MU2 is less tedious from the user's point of view, since `User` does not need to enter $m_{\mathtt{U}}$ using `Mobile`'s input device. The corresponding steps for MU2 are summarized below:

1. `User` $\xrightarrow{\mathtt{Terminal}}$ `Server`: $m_{\mathtt{U}}$;
2. `Server` $\rightarrow$ `Terminal` : $\mathsf{B}(k_s, m_{\mathtt{U}} \| c, v)$;
3. `Terminal` $\rightarrow$ `User`: v;
4. `Terminal` $\rightarrow$ `Mobile`: $\mathsf{B}(k_s, m_{\mathtt{U}} \| c, v)$;
5. `Mobile` $\rightarrow$ `User`: $m_{\mathtt{U}}, c$;
6. `User` $\xrightarrow{\mathtt{Terminal}}$ `Server`: c;
7. `Server` accepts $m_{\mathtt{U}}$ if c is consistent, rejects otherwise.

3.3 Analysis

In this section, we analyze our methods under different adversary models.

Model 1 (`Mobile` is trusted)

In Model 1, we use MU1 for sending message to `Server`, and use MS1 for `Server` to send message to `User` to achieve confidentiality and authenticity of the communication channel.

For both methods, `Terminal` plays the role of a relay point for passing message and thus a malicious `Terminal` is the man-in-the-middle. Hence, this is the classical setting where the two end points (`Server` and `Mobile`) having a shared key want to communicate over a public channel. The cryptographic technique (encryption and message authentication code) can secure the channel and provide both confidentiality and authenticity.

It is clear that MU2 and MS2 cannot protect the confidentiality under this model as the messages are sent in clear through `Terminal`, and thus they are not suitable in this model.

Model 2 (At least one is honest)

In Model 2, we use MU2 to send message to `Server`, and use MS2 for `Server` to send message to `User`. We want to achieve authenticity of the message $m_{\mathtt{S}}$. We are not interested in confidentiality here. It is an interesting future work to investigate whether confidentiality can be achieved under this model.

Suppose `Terminal` is dishonest. In both directions of the communication, we can treat the barcode as the MAC of the message, $m_{\mathtt{U}}$ and $m_{\mathtt{S}}$ respectively. Since `Terminal` does not have the key used in generating the barcode, this is a classical setting and the authenticity of the message inherit from the MAC we used in the barcode construction.

On the other hand, let us consider the case where the `Mobile` is dishonest. In MU2, `Terminal` is honest and will forward $m_{\mathtt{U}}$ to `Server` as it is, thus, it is impossible for `Mobile` to modify $m_{\mathtt{U}}$ without `Server` notices. Similarly, in MS 2, since the actual message $m_{\mathtt{S}}$ is displayed by the honest `Terminal`, `User` can compare the displayed message and thus any modification can be detected.

Note that MU1 and MS1 is not secure in this model: if `Mobile` is dishonest and change the message to m', there is no way for `User` or `Server` to verify it.

Model 3 (No collusion)

It turns out that the protocol we used in method 2, i.e. MU2 and MS2, can achieve authenticity in this model as well.

Let us first analyze MU2. Recall that the goal of a dishonest `Terminal` is to trick `Server` to accept a message $m'_{\mathtt{U}}$. To do so `Terminal` must send `Server` the message $m'_{\mathtt{U}}$, and obtain a barcode b contains $m'_{\mathtt{U}}$ and c. `Server` accepts $m'_{\mathtt{U}}$ only if the verification code c is presented. Since c is randomly chosen, `Terminal` is unlikely to succeed in guessing c. Therefore, he needs to get c from user. Without any hint from `Terminal`, `Mobile` is not able to display the message that the user is expecting.

Now let us analyze MS 2. In this case the dishonest `Terminal` wants to trick `User` into accepting a message $m'_{\mathtt{S}}$. To achieve the goal, it must display $m'_{\mathtt{S}}$ side-by-side with the barcode. As `Terminal` does not know the key k_s he is unable to forge the barcode. Now, consider the dishonest `Mobile`. Recall that there is no communication from the `Terminal` to `Mobile`, the `Mobile` is unable to display the message $m'_{\mathtt{S}}$ which is required to trick `User` to accept $m'_{\mathtt{S}}$.

Table 2 summarizes the security and user friendliness of our methods under different models.

4. VISUAL CHANNEL

A main component in building our visual channel is the construction of 2D barcode with visual cues: given a secret key $k_s = (k_{\mathtt{T}}, k_{\mathtt{E}}, k_{\mathtt{V}})$, a message m, and a visual cue symbol v we want to produce a 2D barcode $\mathsf{B}(k_s, m, v)$ such that the cue v is clearly visible, and the message m can be extracted under noise. On the other hand, there are security requirements on the confidentiality of m and integrity of m and v. Any modification on m and v must be detected with high probability.

4.1 Construction Overview

There are a number of stages in constructing the visual channel:

1. (Encryption-then-MAC): Given m, and the keys $k_{\mathtt{E}}, k_{\mathtt{T}}$, the message m is protected using encryption and MAC with key $k_{\mathtt{E}}$ and $k_{\mathtt{T}}$ respectively, and get $m_0 = \mathcal{E}_{k_{\mathtt{E}}}(m_{\mathtt{U}}) \| \mathcal{T}_{k_{\mathtt{T}}}(\mathcal{E}_{k_{\mathtt{E}}}(m_{\mathtt{U}}))$.
2. (Error correcting): Error correcting code is then applied on the result m_0, and get $\mathsf{ECC}(m_0)$, let us call this m_1.
3. (Embedding visual cue): Given a message m_1, a key $k_{\mathtt{V}}$, and a visual cue v represented as a 2D array of bits, the m_1 is embedded into a larger 2D array of bits I which visually appear as v, Section 4.2 gives details on the embedding process.
4. (Adding control point and rendering): A set of control points(red dots in Figure 1(b)) is then added around I for image registration purpose.

Thus, our barcode is a black and white image with red pixels.

4.2 Encoding with Visual Cue

When a message is too large, multiple barcodes are required to encode it. As mentioned in the introduction, multiple barcodes protected by a single session key are subjected to "rearrangement" attack. To detect the attack, we propose binding location information to the barcode using visual cue. This section gives a method in embedding the visual cue. Note that the process of embedding a visual cue to a barcode can be viewed as the embedding process in digital watermarking, where the visual cue is the host, and the barcode is a message to be "watermarked" to the host.

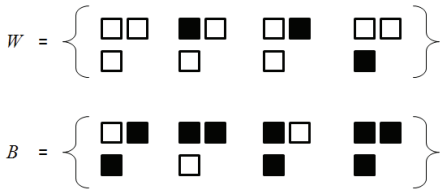
Given a n-bits message m_1, let us arrange it as a x by y binary matrix where $n = x \cdot y$ and x is even. Let us assume that the given visual cue is a $x/2$ by y pixels image where each pixel is either 0 (representing a black pixel) or 1 (representing a white pixel). Therefore, every 2 bits in m is associated with 1 pixel of the visual cue, and together they can be represented with 3 black-and-white pixels in the final barcode. The 3 pixels are arranged in a "L"-shape as shown in Figure 3(a). Let us call the 3 pixels as a L-block. The 2^3 combination of values in a L-block is divided into two groups: W and B. The L-blocks in W have more white pixels and thus the L-blocks appear as "white". Conversely, the L-blocks in B will appear as "black".

Given a binary value $v_1 \in \{0, 1\}$ of a pixel of the visual cue image, we want to encode two bits $\langle b_1, b_2 \rangle$ into a three

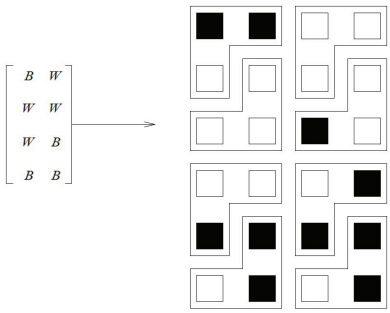
Table 2: Summary of Methods.

	MU 1	MU 2	MS 1	MS 2
Model 1	C, A, U1	A, U1, U2	C, A, U1, U2	A, U1, U2
Model 2	N	A, U1, U2	N	A, U2
Model 3	N	A, U1, U2	N	A, U2

Note: C, A, N are related to security goals and U1, U2 are related to usability.
C: confidentiality is achieved; A: authenticity is achieved; N: none of C and A can be achieved.
U1: no user comparison of messages is required; U2: no user input via `Mobile`'s input device is required.



(a) Two groups of L-blocks.



(b) Tile up with L-blocks.

Figure 3: L-blocks for constructing visual cues

pixels L-block, such that the brightness of the L-block can be adjusted according to v_1. For instance, if $v_1 = 1$, the encoding outputs only elements in W. Since there are 4 elements in W, it is possible to encode the two bits b_1 and b_2. Beside for the value of v_1, there is no further constraint on how the encoding of $\langle b_1, b_2\rangle$ to the 4 elements in W is to be done. In order to prevent the adversary from modifying the appearance of the visual cue, the mapping from the 2 bits $\langle b_1, b_2\rangle$ to the three pixels of the associated L-block, $\langle p_1, p_2, p_3\rangle$, has to be kept secret. Hence, the key space for encoding a bit pair is $4! \times 4! = 576$.

To decode a barcode, `Mobile` applies the decoding and decryption functions in a reverse order and ignore the bit v_1. That is, it first extracts the bit pairs from every L-blocks, and get the message m'. Next, error correcting is applied and the authenticity of the message can be verified.

4.3 Security Analysis

We would like our barcode scheme to achieve the following properties: (1)authenticity and confidentiality of m_{S} and (2) the integrity of visual cue.

Authenticity and confidentiality of message

The authenticity and confidentiality of the message embedded in our barcode scheme rely on the security of the underlying encryption and message authentication scheme. Bellare et al. [2] show that when the encryption $\mathcal{E}_{k_{\mathrm{E}}}$ achieve indistinguishability under chosen-plaintext attack (IND-CPA), and the message authentication scheme $\mathcal{T}_{k_{\mathrm{T}}}$ is strongly unforgeable (SUF-CMA), then the Encrypt-then-MAC composition method achieves IND-CPA, INT-CTXT (integrity of ciphertexts) and IND-CCA ((adaptive) chosen ciphertext attack).

Integrity of visual cue

An adversary may try to modify some L-shape blocks such that the visual cue on two barcode blocks are swapped, and thus, he can rearrange the two blocks without being detected. As discussed in Section 4.2, any modification of an L-shape block's brightness will have $\frac{1}{4}$ chance of not being detected. Suppose at least β number of L-shape blocks have to be modified in order to deceive the user, then the chances of not being detected will be $(\frac{1}{4})^{\beta}$, where β depends on the size of a barcode block, and the visual cue design.

However, the above analysis does not hold when we consider the whole process of decoding, where the error correction is included. Recall that, due to inevitable noise, we need to apply error correcting before extracting $\mathcal{E}_{k_{\mathrm{E}}}(m_{\mathrm{S}})$. Therefore, when small number of L-shape blocks are corrupted, the payload m_1 can still be correctly decoded. Hence, the choice of error-correction and the design of the cues cannot be done separately. Furthermore, some error-correction code can correct more errors than its guaranteed level in some situations. Due to the concern of forgery, it is important not to correct those errors.

To prevent an adversary from making small changes that can deceive the user and yet get verified, one design consideration of the visual cue is to choose symbols with large mutual Hamming distance from each other. In our implementation to be described in Section 7, we use numerical digits as visual cue, where the minimum hamming distance for two symbols is 14 "L-blocks" (for example, the number "1" and "7", "0" and "8"). We choose parameters of error correcting code that is able to tolerate 4 bits noise for every 63 bits. Note that modifying a "L-blocks" may result in two bits flipped, thus, the probability that an attacker can modify the visual cue of a barcode to another is less than $\Phi(3; 14, 0.75) = 3.98\%$ where Φ is the cumulative distribution function of the binomial distribution $\mathbf{B}(14, 0.75)$.

Modifying control points

The adversary may try to modify the control points and this may cause failure in decoding, giving a string of "random" bits which is unlikely to pass the MAC authentication check. Hence, modifications of control points at most amount to a denial of service attack, which is not our main concern.

5. VISUAL CUES FOR VERIFICATION OF MULTIPLE BARCODES

In this section, we discuss a few designs of visual cue, in particular, for barcodes appeared in a linear sequence, and barcodes rendered as table. Recall that the main purpose of the visual cue is to bind location information to the barcodes, so that `User` can visually verify that the barcodes are in the correct arrangement.

Linear Sequential Barcodes.

Consider a sequence of barcodes appearing in the order B_1, $B_2, \ldots, B_n$. The order of appearance gives implicit structure of the encoded message. For instance, the message could be a string divided into substrings where each substring is encoded in a single barcode. Hence, it is important to protect the order of appearance, even if the user may not be interested in viewing all of them. A natural visual cue would be a counter, starting from 1, that is, the visual cue of block B_i is i. To indicate the end of the sequence, the last block contains a special symbol, say "." in our example, to indicate end of sequence.

Barcodes in Table Structure.

Consider a table of messages where each message is encoded in a barcode. The barcodes are depicted in the natural table arrangement: for any 2 messages in the same row, the corresponding barcodes are also in the same row, and likewise for columns. To protect the arrangement, we propose the following rules of assigning the visual cue:

R1 The numerical value of the visual cue symbol on the top row, leftmost block is 1. The value increments by 1 from left to right. At the end of the row, the increment process continues at the leftmost block of the row below if any.

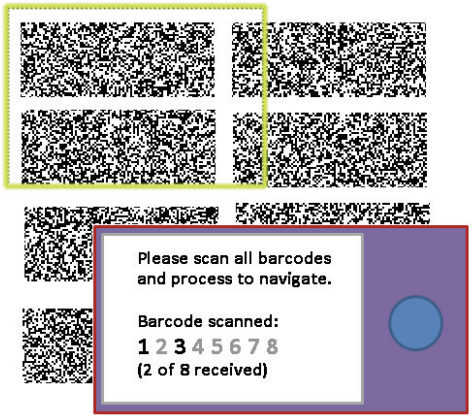
R2 The rightmost block in each row has the additional cue which is a black dot indicating this is the end of row.

R3 The rightmost block in the bottom row has an additional large black rectangle indicating this is the last block.

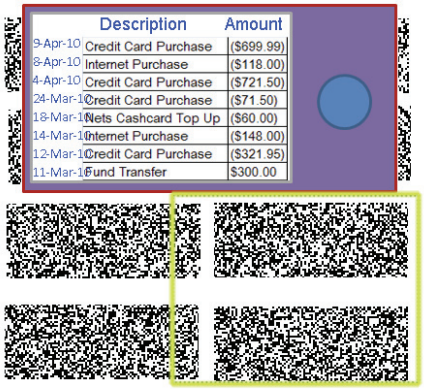
Figure 1(b) shows an example of such barcode table. To verify that a table of barcodes are in the correct arrangement, `User` simply needs to verify the continuity of the counter, every but the last row ends with a small dot, and the last barcode ends with a big dot. It is easy to verify that by imposing the above rules, any insertion, deletion or rearrangement of the barcodes can be detected by visual inspection.

6. ALTERNATIVE METHODS

Besides using visual cues, there are other techniques to ensure that the barcodes are in correct order. This section compares our scheme with a few alternatives. In general, our scheme uses more pixels to carry the visual cue symbols. On the other hand, it has the advantage of requiring less user involvement, incurring less disruption and exploiting the terminal's large display panel. A brief illustration of the alternative methods is given in Figure 4.



(a) `Mobile` captures every blocks, then verifies and renders the whole message.



(b) `Mobile` displays the location(row/column) information encoded in the barcodes.

Figure 4: Illustration of alternative methods (for simplicity, only the barcodes and mobile device are shown here).

Embedding a HMAC of all blocks.

In this method, given a long message m_S, `Server` computes a HMAC for the whole m_S and embeds m_S and its tag into a few barcodes. During authentication, the user first scans across all the barcodes, then `Mobile` responds whether the HMAC agree with the content in the barcodes (Figure 4(a)). If so, `Mobile` renders the long message and user navigates to obtain the required information. The advantages of this method are (1) the user does not need to verify the visual cue, and (2) the barcode is more efficient in the sense that it does not need to embed the visual cue.

However, there are a few disadvantages of this method. Firstly, the scanning process could be less preferred when the user only want to browse a subset of the message (e.g. a user who wants to check a particular record from a list of transactions). Secondly, it is not easy to navigate using the relatively smaller display panel in the mobile device. Furthermore, it is not clear how to extend this method to cater for the setting where `Mobile` is not trusted: one could

display the message in both `Terminal` and `Mobile`, but it is not easy for the user to verify that the displayed messages are consistent when the message is long.

Encoding location hints in barcode.

When the message can be represented as a form of table, one may try to secure the authenticity by using the row and column attributes as location information: Given a table m_s, `Server` first divides it into sub-tables, then it encodes each sub-table together with the corresponding row and column attributes into barcodes. When `Mobile` decodes the barcode, it shows the corresponding attributes of the sub-table as shown in Figure 4(b).

The advantage of this method is that it does not require the user to scan barcodes or verify visual cues, and the user can readily browse a sub-table of interest. While rearrangement attack can be prevented as the row and column information are encoded in the barcode, this method is still subjected to deletion attacks: the adversary may remove or duplicate an entire row of barcode without being detected. Although the "deletion attack" could be patched by encoding more information, for example, by indicating the total number of barcodes, the user is required to be involved in tedious verification, like counting the number of barcode blocks.

7. IMPLEMENTATION

The usability of our proposed method can be improved using "augmented reality" as described in the introduction. We implemented a proof-of-concept system using mobile phones and personal computers.

Deploying Machines and Softwares.

We implemented our method on Android API targeting at OS version v1.6 (Donut), and tested on three mobile devices: (1) a Acer Liquid mobile phone running on Android OS v1.6 with a 3.5 inches 480 × 800 TFT display screen, 256MB RAM, 768 MHz processor, video streaming maximum rate at 20 fps; (2) a Motorola Milestone XT mobile phone running on Android OS v2.1-update1 with a 3.7 in 480 × 854 FWVGA display screen, 256MB RAM, 720 MHz processor and video streaming maximum rate of 24 fps; and (3) a HTC Legend mobile phone running on Android OS v2.1 with 3.2 inches 320 × 480 HVGA display screen, 384 MB RAM, 600 MHz processor, video streaming maximum rate at 30 fps. Let us call these three mobile phones phone 1, phone 2 and phone 3 respectively. We tested the system on three different display units: (1) a 19 inch flat TFT monitor in Dell model Optiplex 755; (2) a 13.3 inch display of a Toshiba portege M900 laptop; and (3) a 15 inch Dell CRT monitor. All configuration of the display units such as brightness resolution are reset to the default setting. Let us call these three display units monitor 1, monitor 2 and monitor 3 respectively. Figure 5 shows an example of our experiment settings.

Choice of Parameters.

We use AES with 128 bit key for encryption scheme, HMAC based on SHA1 for message authentication code, and calculator fonts of numeric digits as visual cues symbols. We use a $(63, 36, 11)$-BCH error correcting code [3] to correct errors. That is, for every 36 bits, we add 27 redundant bits and we are able to correct 5 error bits.

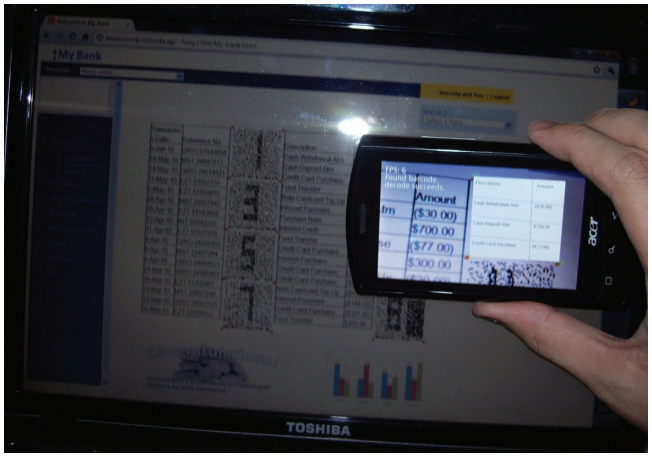


Figure 5: An example of our experiment: browse and verify information on monitor 2 using phone 1.

Image Processing Issues.

We use oversampling technique to reduce the noise of a captured image: one bit in the barcode is rendered using 2×2 pixels. Let us call a group of 2×2 pixels a "superpixel". Such oversampling can reduce the noise due to misalignment and mitigating other artifacts, but it also reduce the channel capacity by a factor of 4. For image registration, each barcode has four 5×5 red dots at the four corners, helping the mobile phone recognize the starting and ending of each barcode. When two barcode are next to each other, we combine the adjacent red dots.

8. PERFORMANCE

In this section we measure the performance of our proof-of-concept implementation in terms of error rate, frame rate and channel capacity.

Error Rate.

We measure the average error rate in reading superpixels with different phones on different monitor. A block 50 by 50 superpixels, together with the 4 red control points are displayed and captured by the mobile device. The errors could be due to motion blur, lens distortion, monitor's refreshing rate, inaccurate image registration, aliasing, and incorrect white balance or focus. Figure 6 shows the result of bit error rates of the barcodes when the four red control points are correctly detected, where the crosses are the bit error rate of a particular captured barcode, the blue plus symbols are the average bit error rates of different phone and monitor, and the red boxes cover the regions between first quartile and third quartile. Although the error rate is affected by the aforementioned factors, Figure 6 shows that the average error rate is acceptable for error correcting to be carried out.

Frame Rate and Decoding Rate.

Our implementation incorporates the framework of "augmented reality": we display the captured video stream as it is, and render the decoded message on top of the video in a separate thread. While the video stream is rendered close to the maximum frame rate of the phone model, the decoding and displaying of message run at a lower rate. The

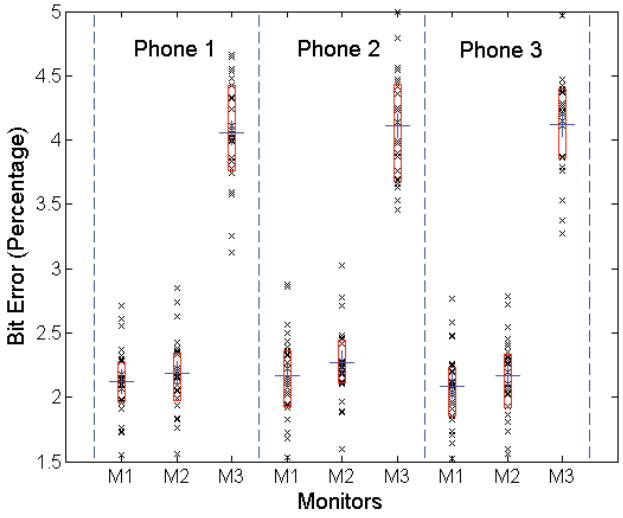


Figure 6: The bit error rate of the three phones capturing barcodes on different monitors.

average decoding rate of our implementation is over 5 cycles per second running on all the three mobile phones.

Capacity of Visual Channel.

We now give calculation for the size of payload (size of m_S, the message `Server` sends to `User`) that can be embedded in a block that occupies 10000 pixels (2500 superpixels) of `Terminal`'s display unit. Recall that we used 2×2 pixels to encode 1 bit of the barcode, employed a $(63, 36, 11)$ BCH error correcting code, and used L-block to preserve the related location. Thus the payload is $10000 \times \frac{1}{4} \times \frac{36}{63} \times \frac{2}{3} = 952$ bits for such a block.

9. RELATED WORK

There is an extensive amount of literatures exploiting the camera as an additional *visual channel* for communication. Jacobs et al. [12] gave a method that establishes a channel from a controllable light source to a camera. McCune et al. proposed seeing-is-believing [16], which carries out authentication and key-exchange over a visual channel established between a device's display and another device's camera. Clarke et al. [5] described a protocol to verify the content on the untrusted terminal by pixel mapping or optical character recognition with a mobile device. Wong et al. [24] built a prototype on a Nokia Series 60 handphone that provides 46 bits for authentication over the visual channel. Sharp et al. [20] gave a system where the sensitive information displayed in the public terminal is blurred or redacted, whereas the mobile device displays the content in the subregion around terminal's mouse pointer in clear. However, the location of the mouse is sent by the terminal and hence potentially a malicious terminal could send the wrong location without begin detected, and thus compromise message integrity. Garriss et al. [10] proposed a protocol that a user can leverage his mobile device to identify, verify Kiosk and submit VM to Kiosk for the user to work on.

Data can be transmitted to a camera effectively using 2D barcodes. There are many 2D barcode designs, for example, QR code [1] and the High Capacity Color Barcode (HCCB) [18] that uses colored triangles. Many barcodes are designed to encode data in printed copies. There are also proposals that use other types of sources in the visual channel. Collomosse et al. proposed "Screen codes" [6] for transferring data from a display to a camera-equipped mobile device, where the data are encoded as a grid of luminosity fluctuation within an arbitrary image. A challenging hurdle in using hand-held cameras to establish the channel is motion blur. A few stabilization algorithms are developed for handheld camera [21, 17], and for 2D barcodes [4].

Similar to our scheme, Costanza et al. [7] suggested a technique to embed designs into barcodes to increase the expressiveness and to bring visually meaning to them. These systems recognize the barcodes based on the topology, rather than geometry, of the codes [8], and were initially developed for tracking objects in tangible user interfaces and augmented reality applications [9]. Augmented reality has been exploited to enhance user experience on many applications including education [14], gaming [22], outdoor activities [23]. Rekimoto et al. [19] Using 2D barcodes as the visual tags in the augmented reality environment, where a camera can capture the barcode on physical object and link them to their information.

10. CONCLUSION

In this paper, we investigated how visual channel can be deployed to enhance security of the communication between server and user in various settings. We pointed out that although authentication of an individual barcode can be easily carried out, the interesting technical challenge is in the verification of the relationships among several barcodes. This leads us to look into the problem of "subregion authentication" where a user wants to verify selective small pieces of data within a large dataset. Although there are a few methods to overcome the problem, they introduce disruptions during the interactive session and are thus less user-friendly. To achieve seamless interactions, we proposed using visual cue to bind location information to the barcode, so as to aid the user in visually verifying the data.

Our protocols demonstrated that, the visual channel "enhanced" with the visual cue, together with the mobile device's input/output device, jointly provide more flexibility in designing secure protocols. Viewing from another perspective, our investigation highlights limitations of visual channel, for instance, the observation that confidentiality is difficult to achieve under the setting where either the mobile device or the terminal could be dishonest. Our solution serves as an interesting example where security is achieved by coupling computer's processing power with human perceptual system. The design of our barcode also serves as an interesting application of fragile watermark.

To illustrate the concept, we implemented the framework, tested on three mobile devices and evaluated it with three types of monitors. The performance of our system is promising and the usability is enhanced with "augmented reality".

Acknowledgement. This work is partially supported by Grant R-252-000-413-232/422/592 from TDSI.

11. REFERENCES

[1] QR Code (2000). International Organization for Standarization: Information Technology-Automatic Identification and Data Capture Techniques-Bar Code Symbology-QR Code. 2000.

[2] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of Cryptology*, pages 469–491, 2008.

[3] R.C. Bose and D.K. Ray-Chaudhuri. On a class of error correcting binary group codes. *Information and control*, pages 68–79, 1960.

[4] C.H. Chu, D.N. Yang, and M.S. Chen. Image stablization for 2d barcode in handheld devices. In *Proceedings of the 15th international conference on Multimedia*, pages 706–715, 2007.

[5] D.E. Clarke, B. Gassend, T. Kotwal, M. Burnside, M. Dijk, S. Devadas, and R.L. Rivest. The untrusted computer problem and camera-based authentication. In *Proceedings of the First International Conference on Pervasive Computing*, pages 114–124, 2002.

[6] J.P. Collomosse and T. Kindberg. Screen codes: visual hyperlinks for displays. In *workshop on Mobile computing systems and applications*, pages 86–90, 2008.

[7] E. Costanza and J. Huang. Designable visual markers. In *Proceedings of the 27th international conference on Human factors in computing systems*, pages 1879–1888, 2009.

[8] E. Costanza and J. Robinson. A region adjacency tree approach to the detection and design of fiducials. *Vision, Video and Graphics*, pages 63–70, 2003.

[9] E. Costanza, S.B. Shelley, and J. Robinson. Introducing audio d-touch: A tangible user interface for music composition and performance. In *Proceedings of the International Conference on Digital Audio Effects*, pages 8–11, 2003.

[10] S. Garriss, R. Cáceres, S. Berger, R. Sailer, L. van Doorn, and X. Zhang. Trustworthy and personalized computing on public kiosks. In *Proceeding of the 6th international conference on Mobile systems, applications, and services*, pages 199–210, 2008.

[11] S. Garriss, R. Sailer, R. Caceres, L. van Doorn, S. Berger, and X. Zhang. Towards trustworthy kiosk computing. In *Workshop on Mobile Computing Systems and Applications*, pages 41–45, 2007.

[12] M.A. Jacobs and M.A. Insero. Method and apparatus for downloading information from a controllable light source to a portable information device, 1996. US Patent 5,535,147.

[13] B. Kauer. OSLO: Improving the security of Trusted Computing. In *Proceedings of 16th USENIX security symposium on usenix security symposium*, pages 1–9, 2007.

[14] E. Klopfer and K. Squire. Environmental detectivesąłthe development of an augmented reality platform for environmental simulations. *Educational Technology Research and Development*, pages 203–228, 2008.

[15] C.Y. Lin and S.F. Chang. Semi-fragile watermarking for authenticating JPEG visual content. In *Proceedings of SPIE*, volume 3971, pages 140–151, 2000.

[16] J.M. McCune, A. Perrig, and M.K. Reiter. Seeing-is-believing: using camera phones for human-verifiable authentication. In *IEEE Symposium on Security and Privacy*, pages 110–124, 2005.

[17] E.M. Or and D. Pundik. Hand motion and image stabilization in hand-held devices. *IEEE Transactions on Consumer Electronics*, pages 1508–1512, 2007.

[18] D. Parikh and G. Jancke. Localization and segmentation of a 2d high capacity color barcode. In *IEEE Workshop on Applications of Computer Vision*, pages 1–6, 2008.

[19] J. Rekimoto and Y. Ayatsuka. Cybercode: designing augmented reality environments with visual tags. In *Proceedings of DARE 2000 on Designing augmented reality environments*, pages 1–10, 2000.

[20] R. Sharp, J. Scott, and A.R. Beresford. Secure mobile computing via public terminals. *Pervasive Computing*, pages 238–253, 2006.

[21] M. Sorel and J. Flusser. Blind restoration of images blurred by complex camera motion and simultaneous recovery of 3d scene structure. In *Signal Processing and Information Technology*, pages 737–742, 2005.

[22] K. Squire and M. Jan. Mad city mystery: Developing scientific argumentation skills with a place-based augmented reality game on handheld computers. *Journal of Science Education and Technology*, pages 5–29, 2007.

[23] G. Takacs, V. Chandrasekhar, N. Gelfand, Y. Xiong, W.C. Chen, T. Bismpigiannis, R. Grzeszczuk, K. Pulli, and B. Girod. Outdoors augmented reality on mobile phone using loxel-based visual feature organization. pages 427–434, 2008.

[24] F.L. Wong and F. Stajano. Multi-channel protocols. In *Security protocols: 13th international workshop*, pages 112–127, 2007.

Exploring Usability Effects of Increasing Security in Click-based Graphical Passwords

Elizabeth Stobert, Alain Forget, Sonia Chiasson,
P.C. van Oorschot, Robert Biddle
Carleton University, Ottawa, Canada
estobert@connect.carleton.ca, aforget@scs.carleton.ca, chiasson@scs.carleton.ca,
paulv@scs.carleton.ca, robert_biddle@carleton.ca

ABSTRACT

Graphical passwords have been proposed to address known problems with traditional text passwords. For example, memorable user-chosen text passwords are predictable, but random system-assigned passwords are difficult to remember. We explore the usability effects of modifying system parameters to increase the security of a click-based graphical password system. Generally, usability tests for graphical passwords have used configurations resulting in password spaces smaller than that of common text passwords. Our two-part lab study compares the effects of varying the number of click-points and the image size, including when different configurations provide comparable password spaces. For comparable spaces, no usability advantage was evident between more click-points, or a larger image. This is contrary to our expectation that larger image size (with fewer click-points) might offer usability advantages over more click-points (with correspondingly smaller images). The results suggest promising opportunities for better matching graphical password system configurations to device constraints, or capabilities of individual users, without degrading usability. For example, more click-points could be used on smartphone displays where larger image sizes are not possible.

1. INTRODUCTION

The problems of knowledge-based authentication, typically text-based passwords, are well known. Users often create memorable passwords that are easy for attackers to guess, but strong system-assigned passwords are difficult for users to remember [25]. Users also tend to reuse passwords across many accounts [17] and this increases the potential impact if one account is compromised. Alternatives such as graphical passwords [4, 26] use images instead of text for authentication. They attempt to leverage the *pictorial superiority effect* [23] which suggests that humans are better able to remember images than text. Some graphical password systems also provide *cueing* [9], whereby a memory retrieval cue is provided to help users remember and distinguish their passwords. In this paper we explore methods to increase the security of cued-recall graphical passwords[1].

We chose to study Persuasive Cued Click-Points (PCCP), a click-based graphical password system in which users select click-points on more than one image [6]. PCCP has been shown to have good usability, while avoiding hotspots that have been shown to affect the security of other click-based graphical password systems [7].

We address the threat of guessing attacks. This danger arises when the total number of possible passwords is small, or when attackers can predict likely passwords. The design of PCCP reduces the predictability of passwords by influencing users during password creation. The number of possible passwords with its standard configuration is 2^{43}, slightly less than that of 7-character random text passwords. A gap in previous literature is that usability tests for graphical password schemes (in general) have only been tested for configurations with password spaces smaller than that of common text passwords. To address this, we explored increasing security in PCCP, conducting a study modifying two parameters: the size of the images presented, and the number of click-points in each password. The study included 82 participants who completed two sessions scheduled two weeks apart. Our results show that both manipulations affect the usability of the system and memorability of the passwords. Moreover, when adjusted to provide the same level of security, both manipulations have similar effects on usability and memorability. This suggests that when increasing security, constraints of devices and user preferences might be accommodated. For example, when designing for mobile devices, smaller images and more click-points might be used due to smaller screen sizes.

The remainder of this paper is organized as follows: we first provide some general background on graphical passwords, and more detail on PCCP. We then introduce our study methodology, and its results. Finally, we discuss the implications of the results and offer our conclusions.

2. BACKGROUND

Graphical password systems [4, 26] are a type of knowledge-based authentication that rely on the human ability to better recognize and remember images than textual or verbal information [23]. They fall into three main categories:

Recall: (also known as drawmetric [11]) Users recall and reproduce a secret drawing on a blank canvas (which may

[1]An early version of part of this work was an extended abstract in the ACM CHI 2010 student research competition.

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

include grid-lines for guidance). Example systems include Draw-A-Secret [20] and Pass-Go [27].

Recognition: (also known as cognometric [11] or search-metric [24]) Users recognize and identify images from a previously memorized portfolio from a larger set of decoy images. Example systems include PassFaces [10] and Déjà Vu [13].

Cued-recall: (also known as locimetric [11]) Users identify and target previously selected locations within one or more images. The images act as memory cues to help recall these locations. Example systems include PassPoints [31] and Persuasive Cued Click-Points [6].

Other approaches to authentication are token-based systems and biometrics. While applicable in some cases, these have potential drawbacks, such as risks of loss, and privacy implications [21]. Password managers have also been proposed, but usability issues and the dangers of centralization remain unsolved problems [8].

In cued-recall click-based graphical passwords [4, 31], passwords consist of clicking on specific locations on one or more images. To log in, the user must click on these previously selected locations. The user is not expected to repeat exact pixel selections. In most systems, an invisible *tolerance square* is defined around each click-point so that any of the enclosed pixels are considered acceptable. Alternatively, a grid may be visible to users [3].

In this paper, we focus on Persuasive Cued Click-Points (PCCP) [6]. In PCCP, a user is presented with a number of images in sequence, and must choose one click-point per image. The first image is assigned by the system, but each subsequent image is determined by the user's previous click. In other words, clicking on different locations on an image results in different next images. This provides users with feedback about the correctness of their password entry attempt — if they see the correct image, they can be fairly certain they have selected the correct click-point on the previous image. However, this *implicit feedback* is not useful to attackers who do not know the correct sequence of images.

Earlier click-based password schemes have a security weakness which makes passwords easier for attackers to predict. Users tend to select similar locations on images, forming *hotspots* [19, 15, 30, 29]. They also tend to select their click-points in predictable geometric patterns [7, 29]. To help create more secure passwords, PCCP includes "persuasive" elements. As shown in Figure 1, the system assists users *only during password creation* by providing a *viewport* that highlights a random part of the image. Users must select a click-point within this viewport. If users are unable to find a memorable point in the current viewport, they may press the *shuffle* button to randomly reposition the viewport. Studies [6, 7] show that this viewport, together with the shuffle button, causes click-points to be more randomly distributed, addressing the predictability problem seen in earlier schemes.

PCCP is stronger against password-guessing attacks than other click-based password systems and also maintains login times and success rates comparable to text passwords [6]. However, to be seriously considered as a replacement for text passwords, PCCP needs to be at least as secure as standard text passwords. We can adjust the security of PCCP by manipulating several parameters, which in turn affect the size of the theoretical password space. However, little research of this nature has been undertaken.

Table 1: Theoretical password space for different text passwords.

Number of Characters	n	Password Space (bits)
95	6	39
95	8	53
95	10	66

Table 2: System parameters for the six experimental conditions and distribution of participants (N).

	w	h	Click-points	Condition Name	Password Space (in bits)	N
Small	451	331	5	S5	44	14
	451	331	6	S6	53	14
	451	331	7	S7	61	14
Large	800	600	5	L5	52	14
	800	600	6	L6	63	12
	800	600	7	L7	73	14

The *theoretical password space* for a password system is the total number of unique passwords that could be generated according to the system specifications. Ideally, a larger theoretical password space lowers the likelihood that any particular password may be guessed. For text passwords, the theoretical password space is typically reported as 95^n, where n is the length of the password, and 95 is the number of typeable characters on the US English keyboard. Table 1 gives the theoretical password space for text passwords of different lengths. For PCCP, the theoretical password space is calculated as: $((w \times h)/t^2)^c$, where the size of the image in pixels ($w \times h$) is divided by the size of a tolerance square (t^2, typically 19^2), to get the total number of tolerance squares per image, then is raised to the power of the number of click-points (c). Table 2 shows the theoretical password space for PCCP with different parameters. As shown in the tables, the theoretical password space for PCCP can be adjusted to approximate the space of text passwords of varying lengths. For example, an 8-character text password has approximately the same password space (2^{53} or 53 bits) as a PCCP password with a small image size (451×331 pixels) and 6 click-points, or a large image size (800×600 pixels) and 5 click-points.

The *effective password space* represents the set of passwords that users are likely to create. For example, in the absence of enforced rules, users of text passwords typically include only lowercase letters, limiting the effective password space to 26^n. For an 8-character password, this would result in a password space of 38 bits. Only rough estimates of the effective password space are available because user choice is based on personal preference rather than mathematical principles. Commonly available text password attack tools such as *John the Ripper* [12] include dictionaries of up to 40 million entries, or 25 bits. Similarly, hotspots and patterns reduce the effective password space in click-based graphical passwords. Since PCCP significantly reduces the occurrence of hotspots and patterns, its effective password space approaches the theoretical password space. By matching the theoretical password space of PCCP to that of text passwords, the corresponding effective password space of PCCP is at least as large (and likely larger) than for text passwords.

Figure 1: User interface for password creation for the small and large image sizes in PCCP.

3. STUDY

Our study explored ways of increasing the password space of PCCP by changing the configuration of the system. With PCCP, three parameters can be manipulated: the image size, the number of click-points per password, and the size of the tolerance square. In this study, we increased the number of click-points in each password and increased the size of the images presented. Our goal was to determine which manipulation resulted in better usability and memorability for approximately equivalent password spaces (as a proxy for security). We chose to keep the size of the tolerance square constant (set to 19×19 as determined in previous studies [31, 5]) because its size is constrained by human visual acuity [16] and fine motor control. We had three hypotheses:

Hypothesis 1(a): Increasing the number of click-points will decrease usability (as defined below).

Hypothesis 1(b): Increasing the size of the image will decrease usability.

Hypothesis 2: For conditions with approximately comparable theoretical password spaces, the condition with the larger image size will have better usability (i.e., L5 would have better usability than S6, and L6 would have better usability than S7).

Our rationale for hypothesis 2 was that conditions with fewer click-points would have better usability because we speculated that the cognitive load and the physical task of entering another click-point would dominate the inspection task of finding a click-point on a larger image.

Our independent variables were the image size and the number of click-points. As shown in Table 2, there were six experimental conditions: *S5* (small image, 5 click-points); *S6* (small image, 6 click-points); *S7* (small image, 7 click-points); *L5* (large image, 5 click-points); *L6* (large image, 6 click-points); and *L7* (large image, 7 click-points). The small image size was 451×331 pixels (the size used in the original PCCP study [6]) and the large image size was 800×600 pixels (standardizing to a 4:3 aspect ratio). These specific settings were chosen to approximate the theoretical password space of text passwords. Our dependent variables concerned usability, and were success rates, duration of password entry, and number of errors. Conditions with shorter durations, fewer errors and higher success rates were judged to have better usability. The level of security was based on the theoretical password space as determined by the independent variables. We also intended to explore the effects of the different conditions on user behaviour in click-point selection, possibly resulting in clustering which reduces the effective password space.

A between-subjects design was used, and the 82 participants (47 females and 35 males) were randomly assigned to one of six study conditions. All participants were regular computer users accustomed to using text passwords. The majority of the participants were university undergraduates, but no participants were studying computer security.

Participants took part in two one-on-one sessions with the experimenter, scheduled approximately two weeks apart. The sessions were 1 hour and 30 minutes long, respectively. Based on previous data, we anticipated that users would be very successful at remembering their passwords during their first session. We had participants wait two weeks before their second session in an effort to counteract ceiling effects and provide measurable differences. Previous studies have shown ceiling effects where participants are extremely successful at remembering their passwords within an hour of creating them, and thus most success rates are close to 100%, providing no measurable differences when in fact differences between conditions may be present.

In their first session, participants initially practiced creating and re-entering passwords for two fictitious accounts, a blog and an online gaming account. This was used to explain the experimental process and familiarize participants with the system. The practice data was discarded and participants did not need to remember these passwords later on. Next, participants created and re-entered PCCP passwords for six fictitious accounts (library, email, bank, online dating, instant messenger, and work). In their second session, participants tried to re-enter these same six passwords.

The experiment used a custom stand-alone J# application running on a Windows desktop computer. A set of 465 images was used, and no images were repeated between or within passwords for a given user. The small and large image conditions shared the same images except that they were displayed at different resolutions. Figure 1 shows the user interface for creating passwords with the two different image sizes. The size of the viewport during password creation was kept consistent at 75×75 pixels across all conditions. Similarly, the tolerance square during all password re-entry phases was 19×19 for all conditions. There were five experiment phases over the two sessions. In the first session, participants completed the *create*, *confirm*, *login* and *recall-1* phases. In the second session, participants com-

Table 3: Success rates on first attempt, within 3 attempts and multiple attempts (eventual success) per phase.

	First Attempt			Within 3 Attempts			Eventual Success		
	Session 1		Session 2	Session 1		Session 2	Session 1		Session 2
Condition	Login	Recall-1	Recall-2	Login	Recall-1	Recall-2	Login	Recall-1	Recall-2
S5	91%	87%	25%	100%	95%	37%	100%	99%	42%
S6	83%	89%	28%	99%	93%	40%	100%	93%	48%
S7	92%	85%	18%	99%	91%	32%	100%	96%	42%
L5	91%	82%	18%	100%	94%	33%	100%	94%	45%
L6	94%	93%	18%	98%	97%	27%	100%	100%	36%
L7	92%	82%	5%	100%	96%	14%	100%	100%	36%

pleted the *recall-2* phase, and were debriefed and compensated for their time. Descriptions of the experiment phases are given below. For each of the six accounts:

Create Phase (Session 1): Participants selected points on images to create their password.

Confirm Phase (Session 1): Participants re-entered the same password to make sure they remembered it. They could re-try as many times as necessary and could reset their password if it was forgotten.

Login Phase (Session 1): Participants attempted to log in to the account using the same password. They could re-try as many times as necessary and could reset their password if it was forgotten.

Once the user had created all their passwords:

Recall-1 Phase (Session 1): Participants attempted to log in to each account in a shuffled order. Multiple attempts were allowed and participants could say they had forgotten a password to move to the next account.

Recall-2 Phase (Session 2): Two weeks later, participants attempted to log in to their accounts in the same shuffled order. Multiple attempts were allowed and participants had the option of saying they had forgotten a password to move to the next account.

4. RESULTS

In this section, we report on the effects of the independent variables (number of click-points and image size) on success rates, errors and durations of password entry. We used statistical analysis to determine whether differences in the data were likely to reflect actual differences between conditions or whether these might reasonably have occurred by chance. Specific tests will be described throughout the section as they are reported. In all cases, we regard a value of $p < .05$ as indicating statistical significance. In such cases there is less than a 5% probability that these results occurred by chance. In the tables reporting statistics, results in bold are statistically significant. Several figures in this section show boxplots to illustrate distributions. Boxplots show the median, the inner quartiles (as a box), and the outer quartiles (as whiskers).

We report on each dependent variable individually, assessing each in relation to the two hypotheses. The phases from Session 1 (create, confirm, login, recall-1) provide a measure of usability in the short-term, while Session 2's recall-2 phase provides a measure of usability after two weeks. Results for each hypothesis are summarized at the end of this section.

Since each user had six separate passwords, we aggregated the data by users to ensure independence in the data. For success rates, we tabulated the number of successful password entries per user, giving a number between 0 and 6.

Table 4: Regression tests for success rates for each phase, only the most relevant measure is reported.

	First Attempt		Within 3 Attempts
	Session 1		Session 2
	Login	Recall-1	Recall-2
Number of Click-points	$p = 0.906$	$p = 0.762$	$\mathbf{p = 0.043}$
Image Size	$p = 0.914$	$p = 0.643$	$\mathbf{p = 0.017}$

For durations, we took the mean of successful password entry times for each user. For errors, we again calculated the mean number of errors for successful password entries.

To test hypotheses 1(a) and 1(b), statistical tests evaluating for main effects of number of click-points and image size were necessary. For statistical tests exploring the effect of number of click-points, we created three distributions grouped on the number of click-points and ignoring image size (i.e., one distribution combining S5 and L5 data, one including S6 and L6, and one including S7 and L7). Similarly, to explore the effect of image size, we created two distributions based solely on image size (i.e., one distribution including S5, S6, and S7, and one distribution including L5, L6, and L7).

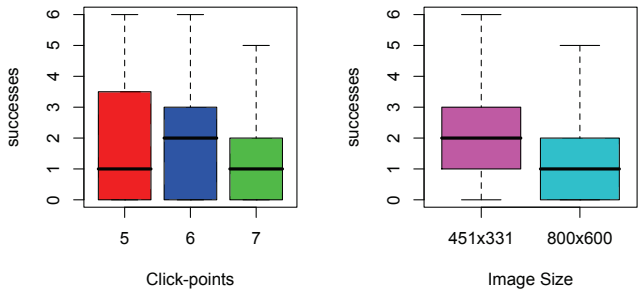


Figure 2: Recall-2 number of successes per user by click-points (left) and by image size (right).

4.1 Success Rates

We report success rates at three different levels: *first time success*, *success within three attempts*, and *eventual success*. First time success occurs when the password is entered correctly on the first attempt, with no mistakes or restarts. Success rates within three attempts indicate that fewer than three mistakes or restarts occurred. Eventual success rates indicate that the participant made multiple attempts, but was eventually successful. Mistakes occur when the participant presses the Login button but the password entry is

Table 5: Mean times in seconds and two-way ANOVA results comparing all 6 conditions for each phase.

	Session 1				Session 2
Condition	Create (s)	Confirm (s)	Login (s)	Recall-1 (s)	Recall-2 (s)
S5	66.9	21.2	16.1	21.5	50.5
S6	109.1	23.3	19.6	20.9	61.5
S7	81.1	28.6	20.8	25.0	75.1
L5	106.2	24.1	18.1	19.3	74.3
L6	103.8	30.2	20.8	23.7	90.5
L7	95.1	32.7	22.0	27.9	81.0
Number of Click-points	$F(2,76) = 0.99$, $p = 0.375$	$\mathbf{F(2,76) = 4.56}$ $\mathbf{p = 0.013}$	$\mathbf{F(2,76) = 5.46}$ $\mathbf{p = 0.006}$	$F(2,76) = 2.40$ $p = 0.097$	$F(2,57) = 0.98$ $p = 0.382$
Image Size	$F(1,76) = 1.68$ $p = 0.200$	$\mathbf{F(1,76) = 4.39}$ $\mathbf{p = 0.039}$	$F(1,76) = 1.73$ $p = 0.193$	$F(1,76) = 0.24$ $p = 0.623$	$F(1,57) = 3.51$ $p = 0.066$

incorrect. Restarts occur when the participant presses the Reset button midway through password entry and restarts password entry. They are analogous to pressing delete while entering text, except that PCCP's implicit feedback helps users detect and correct mistakes during login.

Success rates were examined for the login, recall-1 and recall-2 phases. For hypotheses 1(a) and 1(b), linear regressions were used to look for significant effects of number of click-points and image size. In hypothesis 2, we used Wilcoxon (Mann-Whitney) tests to compare the distributions of the conditions with similar levels of security. Wilcoxon tests are similar to independent sample t-tests, but make no assumptions about the distributions of the compared samples, which is appropriate to the count data in these individual conditions. During the first session (login and recall-1), we consider success on first attempt to be the most important measure of success since users' memory of the password will still be fresh. For recall-2, occurring after two weeks, we consider success within 3 attempts as the most appropriate measure since it most closely reflects account lockout practices for real systems. Results of statistical tests in this section are based these two choices.

Table 3 reports success rates for the login, recall-1 and recall-2 phases. Success rates were very high in Session 1, indicating that participants were very successful at remembering their passwords after a short time period. Success rates after two weeks were much lower, reflecting the difficulty of the memory task. For clarity, Table 3 shows percentages, but the statistical tests were based on the count of successes per user over the six accounts, yielding a number from 0 to 6. Figure 4 shows boxplots indicating the ranges of these counts, distinguishing the different ranges by both click-points and image sizes. Table 4 shows the results of statistical tests using regression to determine whether the differences between the ranges might have occurred by chance.

Hypothesis 1(a): Table 4 shows that in Session 1, neither the login or recall-1 phases showed any significant effects for the number of click-points. For recall-2, there was a significant effect of number of click-points ($p = 0.043$) when considering success within three attempts. This evidence supports hypothesis 1(a) with respect to success rates.

Hypothesis 1(b): As shown in Table 4, varying the image size did not lead to any significant effects in the login or recall-1 phases. In the recall-2 phase, there was a significant effect of image size ($p = 0.017$). This evidence supports hypothesis 1(b) with respect to success rates.

Hypothesis 2: Wilcoxon tests showed no significant differences between S6 and L5 in any phase. Similarly, no significant differences in success rates were found between S7 and L6. Therefore, we have no evidence that having a larger image or more click-points had a larger impact on participants' ability to remember their passwords, offering no support for hypothesis 2.

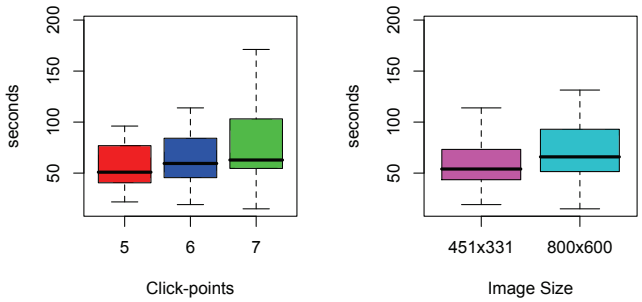


Figure 3: Recall-2 times per user by click-points (left) and by image size (right).

Table 6: t-tests for times: Hypothesis 2

Session	Phase	S6 vs. L5	S7 vs. L6
Session 1	Create	$t(25) = 0.108$ $p = 0.915$	$t(16) = -1.426$ $p = 0.173$
	Confirm	$t(23) = -0.319$ $p = 0.753$	$t(24) = -0.362$ $p = 0.720$
	Login	$t(26) = 1.058$ $p = 0.300$	$t(15) = 0.018$ $p = 0.986$
	Recall-1	$t(14) = 0.851$ $p = 0.409$	$t(21) = 0.303$ $p = 0.765$
Session 2	Recall-2	$t(8) = -0.790$ $p = 0.453$	$t(2) = -0.049$ $p = 0.965$

4.2 Times

Times were measured for each password entry from when the first image appeared on the screen until the participant successfully logged in. This included the time to enter their username, as well as any time making mistakes (pressing the login button and having the system say that the password is incorrect) or resulting from restarts (analogous to pressing the backspace key when entering a text password). All eventually successful password attempts were included

in the time calculations. We ran two-way ANOVAs to examine the main effects of number of click-points and image size. ANOVAs compare variance of the means for multiple samples and identify whether any of the samples are likely to come from different distributions. We used independent samples t-tests to test for significant differences in times between S6 and L5, and between S7 and L6. These tests compare variance of the means between two distributions.

Mean times for each phase are reported in Table 5 and the distributions for recall-2 are seen in Figure 3. No clear pattern emerges in the mean times taken to create passwords, but a general increase in median times can be seen in other phases as more click-points or larger images are used. As should be expected, participants took much longer to re-enter their passwords after two weeks (recall-2), but as intended, this allows comparison between conditions. Table 5 also displays the two-way ANOVA results for main effects of number of click-points and image size.

Hypothesis 1(a): As seen in Table 5, only the confirm and login phases show statistically significant differences for number of click-points. These duration results provide little evidence to support hypothesis 1(a).

Hypothesis 1(b): During recall-2, small increases in median times can be seen in Figure 3 as larger images are used. The only statistically significant effect of image size is seen in the confirm phase. These results offer very little evidence that image size affects time for password entry, and do not support hypothesis 1(b).

Hypothesis 2: As shown in Table 6, no significant differences in durations were seen for S6 vs. L5 or for S7 vs. L6. Participants in conditions with comparable theoretical password spaces could create and recall their passwords equally quickly. We therefore found no evidence to support hypothesis 2 with respect to times.

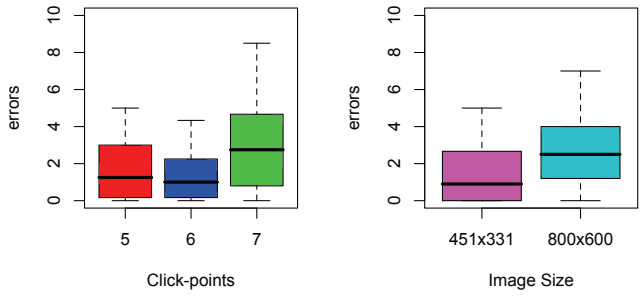


Figure 4: Recall-2 number of errors per user by click-points (left) and by image size (right).

4.3 Errors

An error was recorded every time a participant restarted their password attempt or failed to login because their password was incorrect. Since error distributions were non-normal, we used several non-parametric tests for analysis. When comparing across all conditions, we ran Kruskal-Wallis tests (conventionally reported as χ^2), which are similar to ANOVAs, but used when the distribution of the samples is skewed, as is common with error counts. When comparing two specific conditions, we conducted Wilcoxon (Mann-Whitney) tests to check for significant differences.

Table 7: Mean number of errors per phase.

	Session 1			Session 2
Condition	Confirm	Login	Recall-1	Recall-2
S5	0.43	0.17	0.49	1.33
S6	0.28	0.29	0.05	1.08
S7	0.35	0.11	0.33	2.40
L5	0.45	0.10	0.12	1.79
L6	0.35	0.10	0.17	4.88
L7	0.75	0.10	0.48	4.28

Participants in all conditions made very few errors when entering their passwords during Session 1. For the confirm, login and recall-1 phases, the mean number of errors per account for each phase was less than 1 (Table 7). After two weeks (recall-2), participants made many more errors, as reflected in means ranging between 1.08 and 4.88 errors. This contributed to the longer recall-2 times seen in Section 4.2. The boxplots in Figure 4 show the mean number of errors per user in the recall-2 phase.

Hypothesis 1(a): Kruskal-Wallis tests showed no effect of number of click-points on errors in any phase, therefore offering no support for hypothesis 1(a).

Hypothesis 1(b): In Session 1, increasing the image size had no significant effect on errors. However, there was a significant effect of image size ($\chi^2(1, n = 63) = 8.846, p = 0.003$) in the recall-2 phase, indicating that having larger images caused participants to make more errors after two weeks. This result supports hypothesis 1(b), which stated that increasing image size would decrease usability.

Hypothesis 2: Wilcoxon tests were used to compare the number of errors between S6 and L5 and between S7 and L6. Results showed no significant differences in any phases, providing no evidence to support hypothesis 2.

4.4 Summary of Results

We chose three measures of usability: success rates, times and number of errors. As we describe above, phases from the first session (create, confirm, login, and recall-1) use success on first attempt as the measure of success. Recall-2 uses success within 3-attempts instead. Times and errors include all activity until successful login.

Hypothesis 1(a): *Increasing the number of click-points will decrease usability.* We found partial support for hypothesis 1(a). Although several results indicate a trend towards decreased usability with additional click-points, few statistically significant results were found. The statistically significant differences were in the recall-2 success rates, and in the times taken to confirm and login with passwords.

Hypothesis 1(b): *Increasing the size of the image will decrease usability.* We found evidence supporting hypothesis 1(b). Significant effects of image size were seen in the recall-2 phase for both successes and errors. Users with large images had lower success rates and made more errors than those with small images. A similar trend was seen in recall-2 time results, but statistical tests were not significant.

Hypothesis 2: *For conditions with approximately comparable theoretical password spaces, the condition with the larger image size will have better usability.* There were no significant differences for success rates, times, or number of errors. Our results provide no support for hypothesis 2.

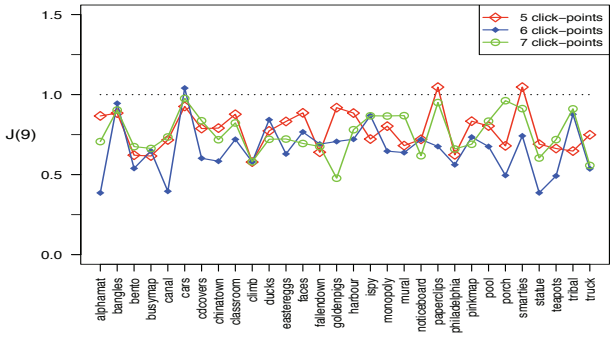


Figure 5: J-statistics for distributions of 5, 6, or 7 click-points. Data from the larger image is scaled to allow for aggregation.

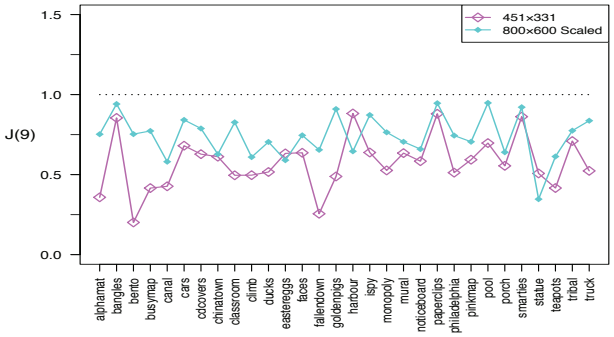


Figure 6: J-statistic for distributions of small and large images. Data from the larger images is scaled to allow generation of comparable J-statistics.

5. CLICK-POINT CLUSTERING

During PCCP password creation, users pressed the shuffle button when they were unable or unwilling to select a click-point within the currently highlighted viewport. We expect fewer shuffles to lead to more randomly distributed passwords, and hence greater security. In this study, there was large variability in the number of shuffles but no clear pattern emerged. The median number of shuffles per password for all conditions is less than five, indicating that most participants pressed the shuffle button less than once per image (passwords consisted of between 5 and 7 images).

Passwords should be as random as possible while still maintaining memorability. Clustering of click-points on an image across users creates what are known as *hotspots*. Attackers who can determine likely hotspots (through image analysis or by gathering a sample of passwords from even a small number of people [30]) would be better positioned to launch an effective dictionary guessing attack. Ideally, a system would minimize the occurrence of hotspots. PCCP attempts to accomplish this through the randomly-positioned viewport, however, users may shuffle the viewport to find a memorable location. We explored whether either image size or number of click-points had an effect on user choice.

To analyze the randomness and clustering of our two-dimensional spatial data, we turned to point pattern analysis [14] commonly used in biology and earth sciences. Our analysis used *spatstat* [2], a spatial statistics package for the *R* programming language.

We used the *J-statistic* [28] as a measure of click-point clustering on a subset of images for which we had sufficient data. Our system ensured that 30 of the images were shown to every participant, giving enough data points for analysis on these particular images. To measure the clustering of points in a dataset, the J-statistic combines nearest-neighbour calculations and empty-space measures for a given radius r. When $J(r) = 0$, it indicates that all points cluster at the same location. When $J(r) = 1$, the points are randomly dispersed across the space. Finally, when $J(r) > 1$, the points are uniformly distributed. For passwords, we want results closer to $J(r) = 1$ since this would be least predictable by attackers. We examined clustering at $J(9)$. A radius of 9 approximates the size of the 19×19 tolerance squares used by our system during password re-entry.

Figure 5 shows the level of clustering for the 30 images, with image names on the x-axis. This figure illustrates the effects of the number of click-points on clustering. Points on each line contain statistics for passwords created using either 5, 6, or 7 click-points. The J-statistic for each image is distinct; the connecting lines are only included for readability. As with earlier analysis in this paper, data from the small (451×331) and large (800×600) images are grouped together based on the number of click-points per password. For example, the 5 click-point line represents all passwords containing 5 click-points regardless of whether they were created on small or large images. The point coordinates on the large images are re-scaled to the coordinate system of the small image so that all data is presented at 451×331 dimensions. This aligns features on the small and large versions of the same images. The lines on the graph do not show any consistent relationship between each other.

To our knowledge, there is no statistical test to compare sets of J-statistics to each other. If we regard the data as categorical, we can identify six categories stemming from the possible orderings: 5-6-7, 5-7-6, 6-5-7, 6-7-5, 7-5-6, 7-6-5. For example, in Figure 5 the *alphamat* image falls in the 5-7-6 category because J(9) for 5 click-points is larger than J(9) for 7 click-points, which is larger than J(9) for 6 click-points. We can then apply a chi-squared test between the observed results and the expected results (equal probability for each category). This test shows no significant differences ($\chi^2(5, n = 60) = 5.675, p = 0.339$). We therefore find no evidence for a difference in clustering between the different numbers of click-points.

Figure 6 shows the level of clustering for the 30 images, distinguishing the effects of image size. Each line contains the statistics for passwords created on either the small or large images. For each of the two cases, data from 5, 6, and 7 click-points are combined. In other words, all passwords created on large images (regardless of how many click-points) are grouped together, and all passwords created on small images (regardless of how many click-points) form a second group. The data from the large images are again scaled to ensure comparability of the J-statistic.

For most images, the graph indicates that the larger images have less clustering (J(9) closer to 1) than the smaller images. If we regard the data as categorical, we could distinguish two categories representing whether the small or large image size has stronger clustering. We applied a chi-squared test between the observed results and the expected results (equal probability for each category). This test shows

a significant difference in clustering for the small and large images ($\chi^2(1, n = 60) = 9.603, p = 0.002$), indicating that larger images have significantly less clustering.

In summary, from Figure 5 it appears that additional click-points do not lead to user behaviour resulting in more clustering. However, larger images appear to influence user choice towards less clustering. This is probably due to the relatively smaller size of the viewport on the larger images. This result suggests that PCCP's shuffle mechanism and viewport (if kept at the same size) is more effective in reducing clustering, and therefore promoting security, when used with larger images.

6. DISCUSSION

We did not see large differences in how the number of click-points and image size affected usability. We expected that increasing the image size would have little or no effect on usability and memorability but we found that it had a similar effect to increasing the number of click-points.

This presents an opportunity, suggesting that other considerations can be taken into account when increasing security. In a situation where choosing a click-point is comparatively difficult (as for a person with a poor fine motor control), this might be accommodated by having fewer click-points, but larger images. More click-points might be appropriate in a situation where screen size was limited, such as on a mobile device. The equivalent demands on usability when increasing security thus give increased flexibility in design.

The conditions under which participants created and used their passwords are clearly artificial. In real life, it is extremely unlikely that a user would create six passwords in a row, then not see them again for two weeks, until they tried to log into all six accounts. The design of our study was meant to emphasize differences between the six conditions by making the task harder. The results of the study for the create, confirm, and login phases are similar to results seen in an earlier study of PCCP [6] and are consistently good, with only small differences between conditions. Further work is needed to confirm real-life usability. We have developed a web-based infrastructure that will allow us to conduct such tests in the near future.

Table 8: Image sizes required, by space and clicks.

Bits	Clicks	X(pixels)	Y(pixels)	X(cm)	Y(cm)
52	6	442	332	11	9
52	5	806	605	21	16
52	4	1986	1489	51	38
52	3	8916	6687	229	171
52	2	179727	134795	4608	3456
62	6	788	591	20	15
62	5	1613	1210	41	31
62	4	4723	3542	121	91
62	3	28305	21229	726	544
62	2	1016688	762516	26069	19552

Multiple passwords are an important issue in authentication. Users typically have many different accounts and are asked to remember many different passwords [17]. This places an increased memory burden on users, and can lead to security and usability problems such as forgetting passwords, and confusing passwords across accounts [18]. Remembering a password for one account can disrupt the memory of a password for another account. This psychological phenomenon is known as *interference* [1]. In our study, participants each created six passwords, each of which was only tenuously linked to a user account. These accounts (library, email, bank, blog, online dating, instant messenger, and work) were denoted only by coloured banners on the login screen (see Figure 1). Although we attempted to emphasize to the user that each account was distinct, there was no practical difference between them. In real life, accounts would be separated from each other by appearance of the website, or created at different times. Participants likely had a hard time distinguishing their passwords from each other, and this interference might have led to more difficulty in remembering them after two weeks.

Although our study focused on several specific configurations of PCCP, it is important to consider the general underlying principles involved.

Image Size: The size of the images shown in each password seems to relate to several human factors. The user likely responds to the appearance of the image with a quick visual survey of the image. While principles of visual attention apply to this survey, the nature of the survey may change with familiarity, or even with exposure to other images or events that relate to the image. The human visual system involves several approaches, including taking in the overall impression, and responding to various attractors. Our initial speculation was that these might be the dominant factors, and we did not expect them to vary much with image size.

For closer inspection of an image, however, the eye will be directed to specific parts of the image. Such close visual inspection requires high acuity vision using the fovea, the area of the retina with a high density of photoreceptor cells [16]. The size of the fovea limits foveal vision to an angle of approximately 1° within the direct line to the target of interest. At a normal viewing distance for a computer screen, say $60cm$, this results in sharp vision over an area of approximately $4cm^2$. The size of the image, and the number of attractors, will then determine the number of foveal areas the user will inspect, and the distance of the saccades as they move from one target to another will also be a factor.

Several factors will affect how PCCP users survey an image. PCCP is a cued-recall scheme, so users will be looking for cues to remind them where to click. PCCP also gives implicit feedback with each image about the previous click, by displaying the correct image if user choose the correct click-point. This means that the user will be assessing whether or not the current image is familiar to them. Then, once users have recognized the image and found their click-point, they must position the cursor correctly using a mouse, touchpad or other pointing device. The time taken to position the cursor may be predicted by Fitts' Law, which determines targeting time from the distance and target size [22]. However, we typically observe users moving the cursor to follow their gaze as they examine the image, so the final movement to a click-point is typically very short.

Click-points: The number of click-points in a PCCP password requires a repetition of all the elements involved in finding and clicking on a single point. We initially assumed this repetition would make the number of click-points a more important factor than the size of the image in determining the usability, but the study results did not support this. In

a pure-recall system, we would expect to see serial memory effects, which cause people to better remember the items at the beginning and end of an ordered list. With PCCP's cued-recall, however, we expect milder serial memory effects, because participants respond to each picture as an individual cue. However, it is certainly possible that users begin to learn the pattern of click-points and anticipate where to focus their gaze, and move their cursor. This anticipation may reduce the work needed per image in ways that have not yet been fully explored.

Table 9: Click numbers required, by space and size.

Bits	X(pixels)	Y(pixels)	Tolerance	Clicks
52	800	600	19	5
52	451	331	19	6
52	320	480	38	8
52	240	320	38	9
52	80	120	38	19
62	800	600	19	6
62	451	331	19	7
62	320	480	38	9
62	240	320	38	11
62	80	120	38	23

Alternative Configurations: It appears that factors such as increasing the number of click-points or image size balance each other out, at least for the settings in our study. To consider the general underlying principles, we might speculate about more extreme possibilities. In our study, the two image sizes used were 451×331 pixels and 800×600 pixels. The tolerance region of the scheme was 19×19 pixels, which meant that the images had approximately 414 and 1330 click areas distinguishable to the system, respectively. Our LCD display measured $43cm$ ($17in$) diagonally with a resolution of 1280×1024 pixels. The small image measured about $12cm \times 9cm$, or $84cm^2$, and the large image about $21cm \times 16cm$ or $336cm^2$. Our study showed that users can cope with inspecting and selecting click-points on images of both sizes within a reasonable amount of time: mean login times were approximately 20 seconds, including entry of username and all click-points.

In our S6 and L5 conditions, the theoretical password space is approximately 52 bits. In S7 and L6, it is about 62 bits. Knowing that the image sizes in these conditions were usable, we explore larger sizes in order to decrease the number of click-points while keeping the password space the same. Table 8 shows some possibilities. For example, even requiring only 3 clicks and keeping the aspect ratio the same would require an image size of 8916×6687 pixels for 52 bits, and 28305×21229 pixels for 62 bits. These would seem to be unreasonable sizes for graphical password images, and would involve a very large number of areas to be inspected. As the number of click-point required decreases, the size of the images implied must grow exponentially, and quickly reaches the bounds of usability. We do navigate on very large *virtual* displays when using cartographic browsers such as Google Earth. This is only manageable, however, through the use of the zoom and pan capabilities, and so the interaction in fact involves a number of clicks.

Implications for Mobile Devices: Our participants managed well with passwords of 5, 6, and 7 click-points in length, so an alternative exploration might be to consider more click points, and allow the image size to be reduced while still maintaining a large password space. Table 9 shows possibilities, using typical small sizes on mobile devices. For example, a small mobile phone might have 120×80 pixels, whereas a Blackberry Curve 8300 has 320×240 pixels, while the Blackberry Bold and the Apple iPhone have 480×320 pixels. Mobile devices sometimes involve a touchscreen instead of a stylus, and often use a dense pixel pitch so images appear physically smaller than the equivalent dimensions on a computer screen. In the table, we accommodate this by using a tolerance region for the mobile devices of 38×38: the size of square onscreen keyboard elements on an iPhone. For the iPhone screen, this would require 8 clicks for a 52 bit password space. These numbers seem potentially acceptable, especially as we frequently type words of that many characters. This suggests that a graphical password scheme such as PCCP might be usable on mobile devices. The small screens will not be compatible with the current viewport because its current size highlights too much of the image to effectively reduce clustering. We are currently exploring a redesigned viewport mechanism. The increasing use of mobile devices for secure online transactions indicates a need for more secure passwords than simple screen unlock mechanisms, and we believe a system such as PCCP has potential for both usability and security.

7. CONCLUSION

In this paper, we explored the issue of how increasing the security of a click-based graphical password scheme would affect usability and memorability. We tested PCCP with different parameters in order to evaluate its usability when the theoretical password space is increased. We found that increasing the number of click-points or increasing the image size both have usability and memorability impacts. While varying parameters to hold constant the size of the theoretical password space, we found no evidence of differences between configurations varying the number of click-points and image size. Additionally, we explored the effects of number of click-points and image size on user behaviour resulting in clustering of click-points. We found no evidence that the number of click-points had an effect, but it appeared that larger images led to less clustering.

These results have important implications for practical configuration of graphical password schemes in various contexts. For example, the results suggest that for mobile devices with small screens, it might be possible to increase security by using smaller images and more click-points while retaining usability and memorability. Conversely, larger images appear to lead to less clustering, suggesting an issue that should be considered in future research.

8. ACKNOWLEDGMENTS

The second author acknowledges NSERC Postgraduate Scholarship funding. The fourth author is Canada Research Chair in Internet Authentication and Computer Security, and acknowledges NSERC funding of this chair, a Discovery Grant, and a Discovery Accelerator Supplement. The fifth author acknowledges funding of an NSERC Discovery Grant. Partial funding from the NSERC Internetworked Systems Security Network (ISSNet) is also acknowledged.

9. REFERENCES

[1] M. Anderson and J. Neely. Interference and inhibition in memory retrieval. In E. Bjork and R. Bjork, editors, *Handbook of Perception and Cognition*, pages 237–313. Academic Press, 1996.

[2] A. Baddeley and R. Turner. R. Spatstat: An R package for analyzing spatial point patterns. *Journal of Statistical Software*, 12(6):1–42, 2005.

[3] K. Bicakci, M. Yuceel, B. Erdeniz, H. Gurbaslar, and N. Atalay. Graphical Passwords as Browser Extension: Implementation and Usability Study. In *Third IFIP WG 11.11 International Conference on Trust Management*, Purdue University, USA, June 2009.

[4] R. Biddle, S. Chiasson, and P. C. van Oorschot. Graphical passwords: Learning from the first generation. Technical Report TR-09-09, Computer Science, Carleton University, `www.scs.carleton.ca/research/tech_reports`, 2009.

[5] S. Chiasson, R. Biddle, and P. C. van Oorschot. A second look at the usability of click-based graphical passwords. In *3rd Symposium on Usable Privacy and Security (SOUPS)*, July 2007.

[6] S. Chiasson, A. Forget, R. Biddle, and P. C. van Oorschot. Influencing users towards better passwords: Persuasive Cued Click-Points. In *Human Computer Interaction (HCI), British Computer Society*, 2008.

[7] S. Chiasson, A. Forget, R. Biddle, and P. C. van Oorschot. User interface design affects security: Patterns in click-based graphical passwords. *International Journal of Information Security*, 8(6):387–398, 2009.

[8] S. Chiasson, P. C. van Oorschot, and R. Biddle. A usability study and critique of two password managers. In *15th USENIX Security Symposium*. Usenix, August 2006.

[9] R. G. Crowder and R. L. Greene. Serial Learning: Cognition and Behaviour. In E. Tulving and F. I. Craik, editors, *The Oxford Handbook of Memory*, chapter 8. Oxford University Press, 2000.

[10] D. Davis, F. Monrose, and M. Reiter. On user choice in graphical password schemes. In *13th USENIX Security Symposium*, August 2004.

[11] A. De Angeli, L. Coventry, G. Johnson, and K. Renaud. Is a picture really worth a thousand words? Exploring the feasibility of graphical authentication systems. *International Journal of Human-Computer Studies*, 63(1-2):128–152, 2005.

[12] S. Designer. John the Ripper password cracker. `http://www.openwall.com/john/`.

[13] R. Dhamija and A. Perrig. Déjà Vu: A user study using images for authentication. In *9th USENIX Security Symposium*, August 2000.

[14] P. Diggle. *Statistical Analysis of Spatial Point Patterns*. Academic Press: New York, NY, 1983.

[15] A. Dirik, N. Menon, and J. Birget. Modeling user choice in the Passpoints graphical password scheme. In *3rd ACM Conference on Symposium on Usable Privacy and Security (SOUPS)*, July 2007.

[16] A. Duchowski. *Eye Tracking Methodology: Theory and Practice*. Springer, 2nd edition, 2007.

[17] D. Florencio and C. Herley. A large-scale study of WWW password habits. In *16th ACM International World Wide Web Conference (WWW)*, May 2007.

[18] S. Gaw and E. Felten. Password management strategies for online accounts. In *2nd Symposium On Usable Privacy and Security (SOUPS)*, July 2006.

[19] K. Golofit. Click passwords under investigation. In *12th European Symposium On Research In Computer Security (ESORICS), LNCS 4734*, September 2007.

[20] I. Jermyn, A. Mayer, F. Monrose, M. Reiter, and A. Rubin. The design and analysis of graphical passwords. In *8th USENIX Security Symposium*, August 1999.

[21] L. Jones, A. Anton, and J. Earp. Towards understanding user perceptions of authentication technologies. In *ACM Workshop on Privacy in Electronic Society*, 2007.

[22] I. S. MacKenzie. Fitts' law as a research and design tool in human-computer interaction. *Human-Computer Interaction*, 7(1):91–139, 1992.

[23] D. Nelson, V. Reed, and J. Walling. Pictorial Superiority Effect. *Journal of Experimental Psychology: Human Learning and Memory*, 2(5):523–528, 1976.

[24] K. Renaud. Guidelines for designing graphical authentication mechanism interfaces. *International Journal of Information and Computer Security*, 3(1):60 – 85, June 2009.

[25] M. A. Sasse, S. Brostoff, and D. Weirich. Transforming the 'weakest link' – a human/computer interaction approach to usable and effective security. *BT Technology Journal*, 19(3):122–131, July 2001.

[26] X. Suo, Y. Zhu, and G. Owen. Graphical passwords: A survey. In *Annual Computer Security Applications Conference (ACSAC)*, December 2005.

[27] H. Tao and C. Adams. Pass-Go: A proposal to improve the usability of graphical passwords. *International Journal of Network Security*, 7(2):273–292, 2008.

[28] M. van Lieshout and A. Baddeley. A nonparametric measure of spatial interaction in point patterns. *Statistica Neerlandica*, 50(3):344–361, 1996.

[29] P. C. van Oorschot, A. Salehi-Abari, and J. Thorpe. Purely automated attacks on passpoints-style graphical passwords. *IEEE Trans. Info. Forensics and Security*, 5(9):393–405, 2010.

[30] P. C. van Oorschot and J. Thorpe. Exploiting predictability in click-based graphical passwords. *Journal of Computer Security*, to appear, 2011.

[31] S. Wiedenbeck, J. Waters, J.-C. Birget, A. Brodskiy, and N. Memon. Authentication using graphical passwords: Effects of tolerance and image choice. In *1st Symposium on Usable Privacy and Security (SOUPS)*, July 2005.

Security Analysis of a Fingerprint-protected USB Drive

Benjamin Rodes
Department of Computer Science
James Madison University
Harrisonburg, VA 22807
benjaminrodes@gmail.com

Xunhua Wang*
Department of Computer Science
James Madison University
Harrisonburg, VA 22807
wangxx@jmu.edu

ABSTRACT

Fingerprint-protected Universal Serial Bus (USB) drives have seen increasing deployment recently to protect mobile data. Compared to regular USB drives, a fingerprint-protected USB drive has an integrated optical scanner and a *private* partition/drive (for example, drive G: on MS Windows), which is *not* accessible before a successful fingerprint authentication.

This paper studies the security of a representative fingerprint-protected USB drive called AliceFDrive. Our results are twofold. First, through black-box reverse engineering and manipulation of binary code in a DLL, we bypassed AliceFDrive's fingerprint authentication and accessed the private drive without actually presenting a valid fingerprint. This authentication bypass is a class attack in that the modified DLL can be distributed to any naive users to bypass AliceFDrive's fingerprint authentication.

Second, in our security analysis of AliceFDrive, we developed a program to *automatically* recover fingerprint reference templates from AliceFDrive, which may make AliceFDrive worse than a regular USB drive: when Alice loses her fingerprint-protected USB drive, she does not only lose her data, she also loses her good-quality fingerprints, which are hard to recover as Alice's fingerprints do not change much over a long period of time.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access controls, authentication, cryptographic controls*; K.6.3 [**Management of Computing and Information Systems**]: Security and Protection—*Authentication, unauthorized access*

General Terms

Security

*Corresponding author. Send all correspondence to wangxx@jmu.edu.

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

Keywords

fingerprint authentication, fuzzy vault, USB drive

1. INTRODUCTION

There have been several highly publicized security breaches of data on mobile storage devices, including the theft of a portable hard disk, owned by a US Department of Veteran Affairs employee, that had confidential data of about 26.5 millions of people [Electronic Privacy Information Center(2006)]. As a result, people have rushed to various security solutions for mobile data, including fingerprint-protected Universal Serial Bus (USB) flash drives. For example, the National Institutes of Health (NIH) has been providing its employees fingerprint-protected USB drives to protect mobile clinical trial data.

A fingerprint-protected USB flash drive (called fingerprint USB drive hereafter) looks like a regular USB drive, except that it has an integrated fingerprint scanner. When a fingerprint USB drive is plugged into a computer running MS Windows, a new read-only *default drive* (for example, drive F:) will appear, on which a program can be found. This program may be automatically run to, among several other things, request fingerprint-based user authentication. (When the fingerprint-protected USB drive is used the first time, its owner can enroll one or more fingerprints.) If the fingerprint authentication succeeds, a new *private* drive/partition (for example, drive G:) will show up, on which the owner can write and read data. This new private drive will *not* be accessible if the fingerprint authentication fails.

A fingerprint USB drive has good usability as the owner does *not* have to remember any reusable passwords. If the drive is stolen/lost, supposedly, an adversary will still *not* be able to read the data on the private drive, due to the lack of appropriate fingerprints.

As a biometric, fingerprint authentication has been first used by government systems and commercial systems that require high-level security (such as a nuclear plant). It has also appeared in many popular Hollywood spy movies such as The Bourne Identity, The Bourne Ultimatum, and Enemy of the State. This history might give ordinary users a perception that fingerprint-protected USB drives offer a high-level security.

In this paper, we explore the following questions: does a fingerprint-protected USB drive really provide better security than a regular USB drive? How hard is it to break the fingerprint protection? Our study is performed on a representative fingerprint-protected USB drive, called *AliceFDrive*[2] — standing for Alice's Fingerprint Drive — through-

out this paper. This pseudonym is used to avoid identifying a specific fingerprint USB drive manufacturer or brand name, as it is not our intention to malign the particular fingerprint USB manufacturer and brand that we tested. AliceFDrive is chosen randomly from its many peers on the market for no particular reason.

Our security analysis focuses on AliceFDrive's software. We treat the AliceFDrive's hardware as a black box and leave it untouched. Unlike a hardware vulnerability, a software vulnerability is suspectable to *class attacks*: software-based exploits for a software vulnerability can be downloaded and used by naive users without knowledge of the vulnerability or the exploit; hence, they are more severe. Our study uses publicly available information only, including AliceFDrive's user's manual, disassembler and debugging toolkits IDA Pro 4.9 [Hex-Rays(2007)] and Ollydbg 1.10 [Yuschuk(2004)], and documents accessible on the web.

The discoveries of this study on AliceFDrive are twofold. First, we demonstrate that, contrary to our initial expectations, it is straightforward to modify the binary code of AliceFDrive to bypass its fingerprint authentication and access its private drive. This modified binary code can be used by other naive AliceFDrive users to circumvent AliceFDrive's fingerprint authentication. Second, we first reverse engineered the format of AliceFDrive's fingerprint minutia points and then developed a program to *automatically* retrieve fingerprint minutia templates from AliceFDrive. From these fingerprint minutia templates, using fingerprint recovery techniques from the research community [Cappelli et al.(2007)Cappelli, Lumini, Maio, and Maltoni, Cappelli et al.(2006)Cappelli, Lumini, Maio, and Maltoni, Hill(2001), Blomme(2003)], we could reconstruct Alice's fingerprints. This may make fingerprint USB drives worse than regular USB drives: when AliceFDrive is stolen, Alice does not only lose her data on the drive, she also loses her good-quality fingerprints, which are hard to recover as her fingerprints remain unchanged in a long period.

The remainder of this article is organized as follows. In Section 2, we give some background information on fingerprint authentication. In Section 3 we describe, from a user's perspective, AliceFDrive and give our initial analysis. In Section 4, we describe our security analysis on AliceFDrive, including methods to bypass the fingerprint authentication, the fingerprint minutia format used by AliceFDrive, and details on recovering fingerprint minutia templates from AliceFDrive. Further discussions on the security of AliceFDrive are given in Section 5. Concluding remarks are given in Section 6.

2. BACKGROUND ON FINGERPRINT AUTHENTICATION

Human fingers have *friction ridges* that are necessary for hands to hold objects firmly [Maltoni et al.(2009)Maltoni, Maio, Jain, and Prabhakar]. The spaces between ridges are called *valleys*. These friction ridges and valleys form *fingerprints* and it is believed that fingerprints exhibit individual patterns for both identification and entity authentication.

There are several levels of fingerprint patterns, namely *global*, *local*, and *very fine* levels. At the global level, fingerprints are categorized in terms of their overall shape, including left loop, right loop, whorl, arch and tented arch. Global level fingerprint characteristics are often not sufficient to differentiate people. Local level fingerprint characteristics consider minute ridge details called *minutia points*, including ridge ending, bifurcation, lake, independent ridge, point or island, spur, and crossover. Among these minutia points, *ridge ending* and *bifurcation* are the most popular and can be well captured by most scanners on the market. Further fine-grained ridge details, including sweat pores, skin creases, scars, and others, have also potential for identification and authentication but they require very high-quality scanner to capture. Minutia-based fingerprints are the most popular these days.

In a fingerprint authentication system, Alice first enrolls her fingerprint with an *authentication server*, which captures Alice's fingerprints, extracts the minutia points, generates a *reference template* and stores it. Later, when an entity wants to be authenticated as Alice, a fresh fingerprint image is captured and it is compared by the authentication server against the stored reference template. Various minutia point-based fingerprint matching algorithms have been developed and they are often threshold based: the fresh fingerprint image is considered genuine as long as it has a threshold or more common minutia points with the reference template. As a result, unlike password or cryptographic key-based comparison, fingerprint matching is close, not exact, matching. The selection of a threshold value depends on the security requirement level.

3. ALICEFDRIVE

3.1 A user's perspective

AliceFDrive comes with a user's manual that describes, from an end user's perspective, AliceFDrive's features. AliceFDrive's main feature is to protect *files* stored on the private drive/partition. In addition to regular files, Alice can also store both favorite web site URLs (i.e., the *favorites* in Internet Explorer and the *bookmarks* in Firefox) and confidential website log-in information (such as user names and passwords) to AliceFDrive. Access to these files on AliceFDrive requires fingerprint authentication.

When AliceFDrive is plugged for the first time into a computer running MS Windows, a program automatically runs and it prompts Alice to enroll her fingerprints. Alice can enroll up to ten fingerprints.

After one or more fingerprints are enrolled, when AliceFDrive is plugged into a computer, two new drives appear, a read-only default drive that contains one executable and a public USB drive. (This public drive acts like regular USB drive and is not discussed in Section 1, as it is *not* security-sensitive.) The executable on the default drive will automatically run for fingerprint authentication. If fingerprint authentication succeeds, the public drive will disappear and a private drive containing confidential data will show up.

Alice can back up her enrolled fingerprints and other user credentials to a file.

Other than claiming AES-256 based "fingerprint encryption," AliceFDrive's user manual does *not* provide details about its security design and implementation.

3.2 Structure of authentication programs

[2]The name of Alice follows the traditional setting of communication security, where Alice is always the message sender and Bob is the receiver. For USB storage security, only one party, Alice, is involved.

The read-only default drive of AliceFDrive has only one program, *AutoRun.exe*, which automatically runs when AliceFDrive is plugged. On Windows XP, an observance on this program shows that *AutoRun.exe* copies some authentication programs, from an unknown source, to *C:\Documents and Settings\All Users\Application Data\AliceFDrive*. These fingerprint authentication programs include one executable (*AliceFDrive.exe*, 2716 kilobytes) and five Dynamic Link Library (DLL) files:
PTSDK4_SS500A_PTFV.dll (32 kilobytes), *PTFVLib.dll* (20 kilobytes), *LTTS1NDUT176.dll* (912 kilobytes), *LTTUSB.dll* (232 kilobytes), and *PasswordBank.dll* (372 kilobytes).

Both the executable and the DLLs are called a *module*. The first step of our security analysis is to find the calling relationship among these modules.

3.2.1 Static module analysis

With IDA Pro [Hex-Rays(2007)], we observed that *AliceFDrive.exe* imports some functions from *PTSDK4_SS500A_PTFV.dll* and the names of these functions start with *bAPI4*. For example, there are two functions *bAPI4_HMFVEnroll* and *bAPI4_HMFVVerify*. (bAPI may stand for biometric Application Programming Interface (API) [ANSI/INCITS(2002)].)

AliceFDrive.exe also imports some functions from *PasswordBank.dll*, with function names such as *iGetOpenIE*, *iOpenUrl*, and *iSaveFormData*. These names suggest that PasswordBank.dll is responsible for storing mobile URL favorites and user passwords.

Further analysis with IDA Pro indicates that *PTSDK4_SS500A_PTFV.dll* statically imports some functions from *PTFVLib.dll* and some other functions, through dynamic loading, from *LTTS1NDUT176.dll*. Also through dynamic loading, *LTTS1NDUT176.dll* imports some functions from *LTTUSB.dll*.

In AliceFDrive, the functions imported by one module from another tend to have meaningful names, such as *bAPI4_HMFVEnroll*. These function names suggest their purposes. However, the types of these functions' parameters are *not* known and without them, one would have to study assembly code to understand how AliceFDrive's functions/modules are implemented. This can be a huge task, as AliceFDrive's modules/functions are fairly complex. For example, *AliceFDrive.exe* has 2845 internal functions – functions defined and used in the same module, *PTFVLib.dll* has 161 internal functions, *LTTS1NDUT176.dll* has 550 internal functions, and *LTTUSB.dll* has 273 internal functions. In other words, without parameter types, it would be very hard to study AliceFDrive's security mechanisms.

3.2.2 Using Google

We next used help from Google by searching those meaningful function names such as *bAPI4_HMFVEnroll*. Google only returned two results and we ended up with a Programmer's Guide for Fingerprint's SDK [Wison Technology Corp.(2009)]. This manual describes a product, Wison Technology OR 200 Optical Sensor and this name does *not* match AliceFDrive. However, the functions described in the manual bear the same names as those exported from *PTSDK4_SS500A_PTFV.dll*. For example, this manual describes *bAPI4_HMFVEnroll*, *bAPI4_HMFVVerify*, and *bAPI4_GetImage* as

```
bAPI4_HMFVEnroll(int iResolution, int iWidth, int iHeight,
            BYTE * pFingerImage, BYTE *pEnrolledFeatures,
            DWORD *pwEnRetSize, int *piStatus)

bAPI4_HMFVVerify(int iResolution, int iWidth, int iHeight,
            BYTE *pFingerImage, BYTE **ppEnrolledfeatures,
            int iEnrolledNum, int *piMatchedID,
            int *piStatus)

bAPI4_GetImage (BYTE *picture, int timeout, int iResolution,
            int *piWidth, int *piHeight)
```

This manual also describes that

- *bAPI4_HMFVEnroll* generates, from a given image (stored in buffer *pFingerImage*), a fingerprint reference template (stored in buffer *pEnrolledFeatures*);
- *bAPI4_HMFVVerify* verifies a given image (stored in buffer *pFingerImage*) against a given set of fingerprint reference templates (stored in buffer ppEnrolledFeatures) and returns a matching result. *bAPI4_HMFVVerify* returns 0 if the verification process fails and returns 1 if the process succeeds. The matching result is stored in *piStatus* and a value of 2 indicates success, a value of 1 indicates failure. If a match is indeed found, *piMatchedID* stores the ID of the matched reference template.
- *bAPI4_GetImage* calls the optical scanner to get an image and stores the result in buffer *picture*.

This manual helps our study in two ways. First, with the functions' parameter types, we can now debug the fingerprint authentication code in a more guided manner. Second, it allows us to develop our own code to call AliceFDrive's DLLs directly, which introduces a lot of flexibility in our study.

It is worth noting that the information provided by this manual is not complete. For example, in what format is the image returned by *bAPI4_GetImage*? It is not clear from the manual and we have to reverse engineer that by ourselves.

4. SECURITY ANALYSIS

In our security analysis, we consider the following scenario: Alice loses her AliceFDrive to Bob, who does *not* have any a priori information about Alice's fingerprints. Bob's goal is to access AliceFDrive's private drive without actually presenting a valid fingerprint and if possible, recover Alice's fingerprints from AliceFDrive. Bob may purchase from the market a brand-new fingerprint-protected USB of the same type as AliceFDrive. Such a new USB drive will be called *BobFDrive* and Bob enrolls his own fingerprints on it. Note that BobFDrive has the same software programs as AliceFDrive. In the following description, we use "we" and "Bob" interchangeably.

Our security analysis of AliceFDrive consists of three steps. First, we shall investigate how hard it is to circumvent AliceFDrive's fingerprint authentication. Second, we will figure out the format of AliceFDrive's fingerprint reference templates, if they are stored on AliceFDrive at all. Third, we will try to recover Alice's fingerprints from AliceFDrive.

4.1 Bypassing fingerprint authentication

Function *bAPI4_HMFVVerify* described in Section 3.2.2 takes a fresh fingerprint image and compares it against a set of fingerprint reference templates enrolled earlier. Understandably, this method is likely called before AliceFDrive's private drive is available. One way to bypass AliceFDrive's fingerprint authentication is to modify *bAPI4_HMFVVerify*'s binary code so that, regardless of the given fresh fingerprint sample, this function returns 1, **piStatus* always returns 2, and its matching ID **piMatchedID* always returns 0.

To find the details of the required changes, the following analysis was performed on BobFDrive. Within IDA Pro, we ran *AliceFDrive.exe* and set a breakpoint at *bAPI4_HMFVVerify*. We then stepped into function *bAPI4_HMFVVerify*, which actually calls function *bPTFVVerify ()* of *PTFVLib.dll*.

We ran *AliceFDrive.exe* twice, one with a correct fingerprint and the other with an incorrect fingerprint. We then found that inside *bPTFVVerify ()* of *PTFVLib.dll*, another function is called and after this call, there is a conditional jump (instruction JZ/JE in assembly language, which has opcode value *0x74*) on register EAX. It jumps when the fresh fingerprint is valid and does *not* jump under an invalid fingerprint.

To bypass fingerprint authentication, we modified this conditional jump to an unconditional jump (instruction JMP in assembly language, with opcode value *0xEB*). A test with this modification in memory succeeded. IDA Pro 4.9 does not support changing binary code persistently. We used OllyDbg 1.10 [Yuschuk(2004)] to save the change back to *PTFVLib.dll*. We then tested this modified *PTFVLib.dll* on AliceFDrive on another computer. With the modified DLL, when provided a non-

matching fingerprint, AliceFDrive's authentication program reported an authentication success and mounted the private drive.

A further inspection of function *bPTFVVerify ()* (of *PTFVLib.dll*) reveals that the above change works only when there are more than one enrolled reference templates. If AliceFDrive has only one enrolled reference template, another subroutine is called and its return value (stored in register EAX) is inspected with instruction *TEST EAX, EAX*. When EAX has a non-zero value, the given fresh fingerprint is considered non-matching and the execution path jumps to a new location with instruction "JNZ". To bypass fingerprint authentication for this case, we simply modified "TEST" (with opcode value *0x85*) to "SUB" (with opcode value *0x2B*) and this change worked. We used OllyDbg to save this modification persistently back to *PTFVLib.dll* and the modified *PTFVLib.dll* was tested successfully.

The above binary code modification is a class attack and a user without any knowledge about fingerprint authentication or binary code analysis can use the modified DLL to bypass the fingerprint authentication and access the private drive of an AliceFDrive USB.

This two-byte change of *PTFVLib.dll*, from $0x74$ to $0xEB$ and from $0x85$ to $0x2B$ respectively, surprised us a little bit. AliceFDrive is a security-related product and we did not expect the authentication bypass to be this easy.

4.2 Recovering minutia templates from AliceFDrive

An attack perhaps more serious than the circumvention of fingerprint authentication on AliceFDrive is to retrieve Alice's fingerprints. A person's fingerprints do not change much over a long period of time and thus cannot be simply revoked. If Bob can reconstruct Alice's fingerprints from AliceFDrive, it would be very hard to recover from this attack and this may raise serious security and privacy concerns. (It is controversial whether fingerprints are *fully* secret data [O'Gorman(2003)], as Alice may leave her fingerprints here and there such as on a water bottle and a desk, but it is more in agreement that fingerprints should be kept as private as possible.)

Our initial thought on this attack is encouraged by the format of function *bAPI4_HMFVVerify*, which takes a fresh fingerprint image and a set of reference templates. This implies that the reference templates must exist, in the clear, somewhere in memory. In what format are these reference templates? Can we develop our own program to directly retrieve these reference templates from AliceFDrive?

4.2.1 Determining minutia template format

To reverse engineer the format of the fingerprint minutia used by AliceFDrive, we developed a program to call the *bAPI4_Enroll()* function, which takes a fingerprint picture (stored in buffer *pFingerImage*) and returns the corresponding reference template in *pEnrolledFeatures*. (This method has to be called three consecutive times (with the same or close image) to get a reference template.)

Before we can call *bAPI4_Enroll()*, we need to figure out the format of the image in *pFingerImage*, which is *not* described by the SDK manual of Section 3.2.2. Using IDA Pro, we followed a normal authentication and dumped the image to a file, which has 89600 bytes, and then analyzed the file. The image is not in JPG, BMP or any other popular graphic formats. Since the resolution of the scanner is 280 by 320 and $89600 = 280 \times 320$, we guessed that each pixel is represented by one byte. We then treated the dumped file as a text file and broke it into lines of 280 bytes. The resulting file was opened in a text editor and it looked like a fingerprint. We then inferred that the image is in raw format and this guess was confirmed when we converted it to an image file and viewed it.

Next, we selected a high-quality fingerprint image (see Figure 1), in which minutiae points are clearly identifiable. This fingerprint image comes from [Maio and Maltoni(1997)] and it has been processed with binarization so that it should be easy for any fingerprint enrollment algorithm, including AliceFDrive, to extract reference templates.

Afterward, we created several variants of this high-quality image through some minor changes, such as removing a ridge that has two minutia points and connecting two ridges to change a minutia point. By enrolling these variants, we hope to infer useful information about the format of AliceFDrive's minutia template format.

For example, when we removed a ridge with two minutia points, we noticed that the sixth byte of the reference template (returned by *bAPI4_Enroll()*) decremented by two (from 24 to 22) and the overall size of the reference template decreased by 12 (from 151 to 139). We then inferred that that the sixth byte must be part of a field indicating the number of minutia points in the fingerprint and each minutia point is represented by six (i.e., $\frac{12}{2}$) bytes. This also lets us infer that a reference template has a header of seven bytes (i.e., $(151 - 24 \times 6)$).

For this seven-byte header, the first byte is $0x09$ in most situations and the meaning of the second byte is still unknown to us. The sixth and the seventh bytes together, in the little endian format, denote the number of minutia points in the fingerprint. We will explain the third, fourth and fifth bytes shortly.

For each minutia point, in what format are its six bytes? We checked several standards for minutia point representation, including ANSI/INCITS 378-2004 [ANSI/INCITS(2004)], ANSI/NIST-ITL1-2007 [NIST(2007)], CDEFF [NIST(2009), Podio et al.(2004)Podio, Dunn, Reinert, Tilton, Struif, Herr, et al.], and ISO IEC 19794-2 [ISO/IEC(2005)]. None of them work for AliceFDrive. This made us think that AliceFDrive uses a proprietary format.

Figure 1: A fingerprint picture after binarization. Reprinted with permission from [Maio and Maltoni(1997)]. © 1997 IEEE.

To determine the format of the six bytes for a single minutia point, we followed the execution of *bAPI4_HMFVEnroll*, which is the most labor-intensive step in this study. Our study shows that *bAPI4_HMFVEnroll* first processes, in many complex steps, the given fingerprint image into a global structure. *bAPI4_HMFVEnroll* then processes this global structure into a flat structure and returns it in *pEnrolledFeatures*. In this flattening operation, four variables are packaged to a 32-bit word, where the first 11 bits of the 32-bit word comes from the first variable, the next 11 bits from the second variable, the next 2 bits from the third variable and the remaining 8 bits from the fourth variable. This 32-bit word is then saved in little endian format.

We guessed that these four variables might represent the coordinates, including x and y, the type, and the angle of the minutia point respectively. To verify this guess, we interpreted the reference template returned by *bAPI4_HMFVEnroll* as guessed above

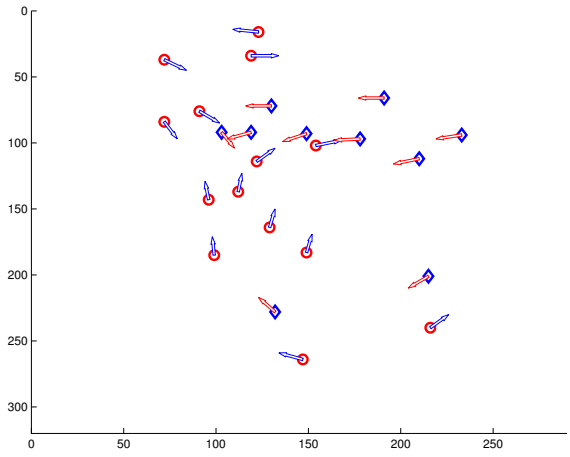


Figure 2: Recovered minutia template

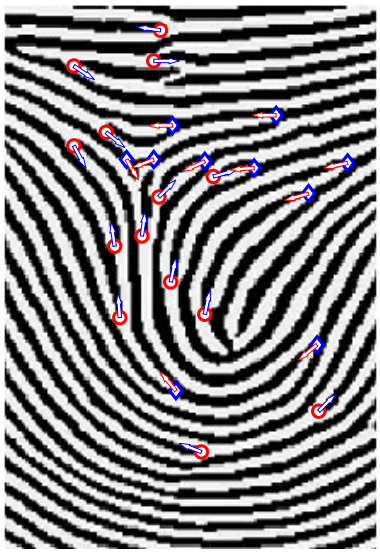


Figure 3: Picture marked with recovered minutia points, after some transformations

to get a set of minutia points and then drew a *figure* with the minutia points in terms of their coordinates (x, y), types and angles (see Figure 2); we next tried to match this figure with the enrolled fingerprint *image* (that is, Figure 1). If our guess is correct, we should be able to superimpose the figure on the original fingerprint image. There was no immediate matching between the figure and the image. However, when the image was rotated 180 degrees and flipped, we could match the figure and the image after three transformations (see below for more details on these transformations). The result is depicted in Figure 3, where red circles denote minutia points of type ridge ending, blue diamonds denote minutia points of type bifurcation, and arrows denote angles of the minutia points.

In Figure 3, three transformations are used in the drawing of minutia points. First, the origin of the natural coordinate system for the minutia points is located at the left bottom and its y axis of the coordinate system grows upward; in contrast, the origin of the image is located at the upper left corner and its y axis grows downward. To superimpose the figure with the image, in Figure 3, the minutia points' y coordinates are adjusted toward the image's coordinate system. Second, in the reference templates returned by *bAPI4_HMFVEnroll*, the coordinates of all minutia points are relative to a fixed point, whose location is determined by the third, the fourth, and the fifth bytes of the seven-byte header of the reference template. More specifically, the third byte is the y coordinate of the fixed point and the x coordinate of the fixed point is stored in the fourth and fifth bytes in little endian format. This relative adjustment is used in generating Figure 3. Third, each minutia point's angle uses one byte and its value ranges from 0 to 255. To represent any degrees between 0 and 359, a multiplication factor of $\frac{360}{256}$ is used when the byte value is interpreted as degrees.

The quality of Figure 3 gives us high confidence that our interpretation of the reference template is correct. Our next task is to study how to retrieve reference templates from AliceFDrive.

4.2.2 Retrieving minutia templates from AliceFDrive

The parameters of *bAPI4_HMFVVerify* imply that AliceFDrive's reference templates exist in the clear in memory and we have confirmed this by running AliceFDrive's authentication program within IDA Pro. However, reading memory of a computer program usually requires some expertise and may be a daunting task for ordinary users. An even better break is to develop a computer program to *automatically* retrieve reference templates from AliceFDrive directly.

In our tracing of AliceFDrive's authentication program, we noticed a call to function *bAPI4_ReadSecureArea* with three parameters $(dst, 40, 0)$, where dst is a buffer of $0x5000$ bytes. After this call, the retrieved data is then processed and immediately afterward, the reference templates appear in cleartext in memory. The processing subroutines look fairly complex but their program structure does look like an Advanced Encryption Standard (AES) T-box-based implementation [Daemen and Rijmen(2002), National Institute of Standards and Technology(2001)], where AES T-box is generated beforehand and is used in decryption. The processing subroutines comprise a subroutine that looks like AES key scheduling and it takes 16 bytes of $0xff$, which might be the AES key.

To verify this observation, we developed a program that first calls *bAPI4_ReadSecureArea* to read data and then decrypts it with AES-128 with 16 bytes of $0xff$ as the key.

The decrypted text consists of several parts: it starts with a 16-byte ASCII string, "AliceFDrv AESKEY", followed by some fields and a set of user profiles, including user names, passwords and their Windows domain names. The decrypted text ends with a binary string, which looks like a set of reference templates. We then interpreted the reference template section with the reference template structure obtained in Section 4.2.1 and the second reference templates is depicted in Figure 4.

To verify our interpretation of minutia points retrieved from AliceFDrive, we talked to Alice and obtained an image of her second enrolled fingerprint. Figure 5 was drawn by superimposing Figure 4 on Alice's fingerprint image with a shifting of $(x = 20, y = 50)$. (This shifting is necessary due to the displacements of Alice's finger in two different scans, namely her enrollment scan and the later scan.)

The high quality of Figure 5 further confirms our interpretation of AliceFDrive's reference templates. Once we have recovered Alice's fingerprint minutia templates, we may be able to reconstruct Alice's fingerprints with existing techniques of fingerprint reconstruction from minutia templates [Galbally et al.(2008)Galbally, Cappelli, Lumini, Maltoni, and Fierrez, Cappelli et al.(2007)Cappelli, Lumini, Maio, and Maltoni, Cappelli et al.(2006)Cappelli, Lumini, Maio, and Maltoni, Ross et al.(2007)Ross, Shah, and Jain]. This will allow us to impersonate Alice for a very long time.

AliceFDrive also supports user profile backup and with the

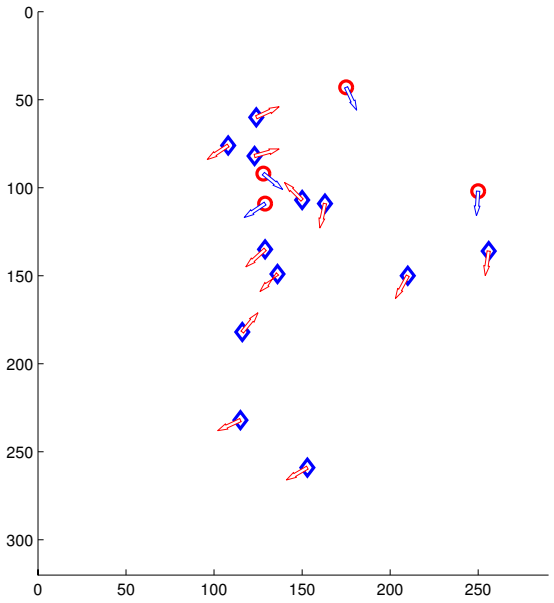


Figure 4: A minutia template retrieved from AliceFDrive

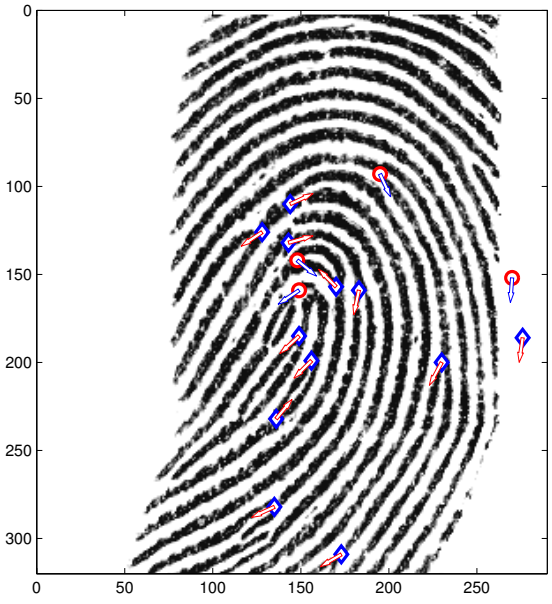


Figure 5: Retrieved minutia template on a fresh fingerprint, after alignment

same techniques described above, we decrypted the backup file and recovered Alice's minutia points from it.

4.3 Summary

In this section, we described two ways to break AliceFDrive. First, through two byte changes to the PTFVLib.DLL, AliceFDrive's fingerprint authentication was bypassed and its private drive was accessed without ever presenting a valid fingerprint. This is a class attack as the modified DLL can be downloaded and used by any naive users to bypass AliceFDrive's fingerprint authentication.

Second, we developed a program that can automatically retrieve reference templates from AliceFDrive. Using existing techniques of fingerprint reconstruction from minutia points, we could reconstruct Alice's fingerprints from a lost AliceFDrive. This poses serious security and privacy concern as now Alice does not only lose her private data, but also her good-quality fingerprints, which remains unchanged for a long period and cannot be simply revoked.

5. DISCUSSIONS

As shown in Section 4, AliceFDrive, as a security product, is highly vulnerable to authentication bypass and its fingerprint minutia templates can be recovered from the drive. In this section, we shall discuss how these vulnerabilities can be fixed or at least, mitigated to improve AliceFDrive's security.

5.1 Program structure and code/API obfuscation

The security analysis of AliceFDrive can be made more difficult in several ways.

First, the structure of AliceFDrive's authentication programs can be made less obvious. Our security analysis was greatly aided by the simple calling relationship among AliceFDrive's six modules (i.e., one executable and five DLLs), the meaningful function names exported by these modules, and the document found by Google on these functions and their parameters.

The location of code of interest for authentication bypass can be made hard to find if the function names are scrambled and the code is obfuscated. Also, anti-debugging techniques can be employed to foil debugging of the authentication programs. This would not stop a committed attacker, but it would increase the time and effort necessary for the attack.

5.2 Using fuzzy vault and some problems

The security improvements discussed in Section 5.1 are heuristic in that they are theoretically breakable: given enough efforts and time, they can always be broken.

An ultimate secure solution to fingerprint-protected USB drive is to derive a *consistent* cryptographic key from close fingerprints and use it to encrypt all data on the private drive [Juels and Sudan(2002), Juels and Sudan(2006)]. In this way, the fingerprint-based authentication can *not* be bypassed: an adversary can manipulate the authentication program's execution path but without a valid fingerprint, the same cryptographic key can *not* be reconstructed and consequently, the encrypted data on the private drive cannot be decrypted. Also, in this solution, *no* plain reference template is stored on AliceFDrive and thus an adversary can *not* reconstruct it from a stolen/lost AliceFDrive.

Since fingerprints captured from the same finger are often close, but not exactly the same, due to a lot of environmental factors (such as the quality of the scanner, moisture, and scratches), the main challenge of this approach is to extract the same cryptographic key from close-but-not-exactly-the-same fingerprints. More specifically, for minutia-based fingerprints that are represented by a *set* of minutia points, how to extract consistent cryptographic keys from close sets?

The concepts of *fuzzy extractor* [Dodis et al.(2004)Dodis, Reyzin, and Smith, Dodis et al.(2008)Dodis, Ostrovsky, Reyzin, and Smith] and *fuzzy vault* [Juels and Sudan(2002), Juels and Sudan(2006)] have been developed to address this problem. There have been a couple of set-based fuzzy extractor/vault constructions, including the one by [Juels and Sudan(2002), Juels and Sudan(2006)] and its improvement by [Dodis et al.(2004)Dodis, Reyzin, and Smith, Dodis et al.(2008)Dodis, Ostrovsky, Reyzin, and Smith], which are based on the set difference metric. A construction based on the set intersection and its improvement can be found in [Socek et al.(2007)Socek, Božović, and Ćulibrk, Wang et al.(2008)Wang, Huff, and Tjaden]. These constructions allow fuzzy comparison of close sets in the "encrypted" form and assume that the comparison of elements in the close sets is *either* exact (i.e., an element

from one set is considered in another close set if and only if the element itself — not a close copy — appears in the second set) *or* very close. (Exact element-level comparison is required when an existing fuzzy vault scheme does not use chaff points and when chaff points are indeed used, element-level comparison should be very close.) These fuzzy extractor/vault schemes provide provable security. However, requiring element-level exact or very-close comparison makes them not practicable for minutia point-based fingerprints. Given two minutia-based fingerprints from the same finger, due to displacement, rotation, and distortion, before alignment, a minutia point in one fingerprint can be pretty far from its matching point in the close fingerprint.

To use these existing fuzzy vault schemes, a fresh fingerprint has to be pre-aligned before it is used by a fuzzy vault scheme. This pre-alignment would require AliceFDrive to store some *helper data* about the fingerprint and this help data may leak information about the fingerprint, making the system less secure or even insecure. In other words, the provable security provided by the original fuzzy extractor/vault scheme is lost. Nandakumar et al. implements such a tradeoff and stores curvature of the fingerprint as helper data [Nandakumar et al.(2007)Nandakumar, Jain, and Pankanti]. The exact security of this implementation remains to be studied.

5.3 Integrating a processor to the fingerprint USB drive

AliceFDrive uses a host computer to run its program for fingerprint enrollment and fingerprint verification, making it susceptible to this security analysis. One natural security improvement is to integrate to AliceFDrive a tamper-resistant microprocessor and have it perform the fingerprint enrollment and verification function. In this way, fingerprint authentication cannot be simply bypassed and fingerprint reference templates can be better protected.

One challenge facing such a microprocessor integration design is that the fingerprint enrollment and verification procedures are often computation-intensive and an embedded microprocessor with limited computation power may not handle these computations very well.

6. CONCLUSION

USB drives with large storage capacity support data mobility and improve productivity. They also pose serious security challenges. Fingerprint-protected USB drives protect a private USB partition with fingerprint authentication and have been increasingly popular. This paper analyzes the security of AliceFDrive, a representative fingerprint-protected USB drive.

In our study, we showed that it is straightforward — just the change of two bytes in a DLL of the authentication programs — to bypass AliceFDrive's fingerprint authentication. This authentication bypass is a class attack, as any naive users can download the modified DLL and bypass AliceFDrive's fingerprint authentication.

In this study, we also developed our own program to *automatically* recover fingerprint reference templates from AliceFDrive. With existing techniques of reconstructing fingerprints from minutia point templates, we could reconstruct Alice's fingerprints and impersonate Alice for a long period of time.

It is our hope that this study will inspire the information security community to search for better solutions to improve the security of fingerprint-protected USB drives.

7. ACKNOWLEDGMENTS

The authors would like to thank Brett Tjaden for reviewing earlier draft of this paper and Florian Buchholz for helpful discussions. We'd also like to thank the anonymous reviewers for constructive comments and for suggesting Section 5.3.

8. REFERENCES

[ANSI/INCITS(2002)] ANSI/INCITS. Information technologoy - BioAPI specification. ANSI/INCITS 358-2002, February 2002. Version 1.1.

[ANSI/INCITS(2004)] ANSI/INCITS. Finger minutiae fomat for data interchange. ANSI/INCITS 378-2004, 2004.

[Blomme(2003)] J. Blomme. Evaluation of biometric security systems against artificial fingers. LITH-ISY-EX-3514-2003, Department of Electrical Engineering, Linköping University, Linköping, Sweden, October 2003.

[Cappelli et al.(2006)Cappelli, Lumini, Maio, and Maltoni] R. Cappelli, A. Lumini, D. Maio, and D. Maltoni. Can fingerprints be reconstructed from ISO templates? In *Proceedings of the International Conference on Control, Automation, Robotics and Vision (ICARCV2006)*, December 2006.

[Cappelli et al.(2007)Cappelli, Lumini, Maio, and Maltoni] R. Cappelli, A. Lumini, D. Maio, and D. Maltoni. Fingerprint image reconstruction from standard templates. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(9):1489 – 1503, Sept. 2007.

[Daemen and Rijmen(2002)] J. Daemen and V. Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002. ISBN 3-540-42580-2.

[Dodis et al.(2004)Dodis, Reyzin, and Smith] Y. Dodis, L. Reyzin, and A. Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In C. Cachin and J. Camenisch, editors, *Advance in Cryptology — EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 523–540, 2004.

[Dodis et al.(2008)Dodis, Ostrovsky, Reyzin, and Smith] Y. Dodis, R. Ostrovsky, L. Reyzin, and A. Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. *SIAM Journal on Computing*, 38(1): 97–139, 2008. URL `http://link.aip.org/link/?SMJ/38/97/1`.

[Electronic Privacy Information Center(2006)] Electronic Privacy Information Center. Veterans affairs data theft, July 2006. URL `http://epic.org/privacy/vatheft/`.

[Galbally et al.(2008)Galbally, Cappelli, Lumini, Maltoni, and Fierrez] J. Galbally, R. Cappelli, A. Lumini, D. Maltoni, and J. Fierrez. Fake fingertip generation from a minutiae template. In *Proceedings of the 2008 International Conference on Pattern Recognition (ICPR08)*, pages 1–4, 2008.

[Hex-Rays(2007)] Hex-Rays. IDA Pro 4.9 freeware, February 11 2007. URL `http://www.hex-rays.com/idapro/idadownfreeware.htm`.

[Hill(2001)] C. Hill. Risk of masquerade arising from the storage of biometrics. BSc Honours Thesis, Department of Computer Science, Australian National University, November 2001.

[ISO/IEC(2005)] ISO/IEC. Information technology - biometric data interchange format - part 2: Finger minutiae data. ISO/IEC 19794-2, 2005.

[Juels and Sudan(2002)] A. Juels and M. Sudan. A fuzzy vault scheme. In *Proceedings of the IEEE International Symposium on Information Theory (ISIT 2002)*, Lausanne, Switzerland, 2002.

[Juels and Sudan(2006)] A. Juels and M. Sudan. A fuzzy vault scheme. *Designs, Codes, and Cryptography*, 38(2):237–257, 2006.

[Maio and Maltoni(1997)] D. Maio and D. Maltoni. Direct gray-scale minutiae detection in fingerprints. *IEEE Transactions on Pattern Analysis and Machine Learning*, 19(1):27–40, January 1997.

[Maltoni et al.(2009)Maltoni, Maio, Jain, and Prabhakar] D. Maltoni, D. Maio, A. K. Jain, and S. Prabhakar. *Handbook of Fingerprint Recognition*. Springer, 2nd edition, 2009. ISBN 978-1-84882-253-5.

[Nandakumar et al.(2007)Nandakumar, Jain, and Pankanti] K. Nandakumar, A. K. Jain, and S. Pankanti. Fingerprint-based fuzzy vault: Implementation and performance. *IEEE Transactions on Information Forensics and Security*, 2(4):744–757, December 2007.

[National Institute of Standards and Technology(2001)]

National Institute of Standards and Technology. Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, 2001. URL http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[NIST(2007)] NIST. Data format for the interchange of fingerprint facial, & other biometric information - part 1. NIST Special Public Report: 500-271, May 2007. URL http://fingerprint.nist.gov/standard/Approved-Std-20070427.pdf.

[NIST(2009)] NIST. Data format for the interchange of extended friction ridge features. Proposed Addendum/Revision to ANSI/NIST-ITL 1-2007, WORKING DRAFT Version 0.4, June 2009. URL http://fingerprint.nist.gov/standard/cdeffs/Docs/CD-EFFS_DraftStd_v04_2009-06-12.pdf.

[O'Gorman(2003)] L. O'Gorman. Comparing passwords, tokens, and biometrics for user authentication. *Proc. IEEE*, 91(12): 2019–2040, Dec. 2003.

[Podio et al.(2004)Podio, Dunn, Reinert, Tilton, Struif, Herr, et al.] F. L. Podio, J. S. Dunn, L. Reinert, C. J. Tilton, B. Struif, F. Herr, J. Russell, M. P. Collier, M. Jerde, L. O'Gorman, and B. Wirtz. CBEFF common biometric exchange formats framework. NISTIR 6529-A, April 5th 2004. URL http://csrc.nist.gov/publications/nistir/NISTIR6529A.pdf.

[Ross et al.(2007)Ross, Shah, and Jain] A. Ross, J. Shah, and A. K. Jain. From template to image: Reconstructing fingerprints from minutiae points. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(4):544–560, April 2007.

[Socek et al.(2007)Socek, Božović, and Ćulibrk] D. Socek, V. Božović, and D. Ćulibrk. Practical secure biometrics using set intersection as a similarity measure. In *Proceedings of International Conference on Security and Cryptography (SECRYPT 2007)*, Barcelona, Spain, July 28-31 2007. INSTICC.

[Wang et al.(2008)Wang, Huff, and Tjaden] X. Wang, P. D. Huff, and B. Tjaden. Improving the efficiency of capture-resistant biometric authentication based on set intersection. In *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC 2008)*, pages 140–149, Anaheim, CA, December 8-12 2008. IEEE Computer Society Press.

[Wison Technology Corp.(2009)] Wison Technology Corp. Programmer's guide for fingerprint's SDK, April 2009. URL http://www.wison.com.tw/cht/document/Wison2/DOC/OR2-00_ProgrammerGuide.pdf.

[Yuschuk(2004)] O. Yuschuk. Ollydbg 1.10. Freeware, 2004. URL http://www.ollydbg.de/.

A Quantitative Analysis of the Insecurity of Embedded Network Devices: Results of a Wide-Area Scan

Ang Cui and Salvatore J. Stolfo
Department of Computer Science, Columbia University
{ang,sal}@cs.columbia.edu

ABSTRACT

We present a quantitative lower bound on the number of vulnerable embedded device on a global scale. Over the past year, we have systematically scanned large portions of the internet to monitor the presence of trivially vulnerable embedded devices. At the time of writing, we have identified over **540,000** publicly accessible embedded devices configured with factory default root passwords. This constitutes over **13%** of all discovered embedded devices.These devices range from enterprise equipment such as firewalls and routers to consumer appliances such as VoIP adapters, cable and IPTV boxes to office equipment such as network printers and video conferencing units. Vulnerable devices were detected in **144 countries**, across 17,427 unique private enterprise, ISP, government, educational, satellite provider as well as residential network environments. Preliminary results from our longitudinal study tracking over 102,000 vulnerable devices revealed that over **96%** of such accessible devices remain vulnerable after a 4-month period. We believe the data presented in this paper provides a conservative lower bound on the actual population of vulnerable devices in the wild. By combining the observed vulnerability distributions and its potential root causes, we propose a set of mitigation strategies and hypothesize about its quantitative impact on reducing the global vulnerable embedded device population. Employing our strategy, we have partnered with Team Cymru to engage key organizations capable of significantly reducing the number of trivially vulnerable embedded devices currently on the internet. As an ongoing longitudinal study, we plan to gather data continuously over the next year in order to quantify the effectiveness of community's cumulative effort to mitigate this pervasive threat.

1. INTRODUCTION

Embedded network devices have become an ubiquitous fixture in the modern home, office as well as in the global communication infrastructure. Routers, NAS appliances, home entertainment appliances, wireless access points, web cams, VoIP appliances, print servers and video conferencing units reside on the same networks as our personal computers and enterprise servers and together form our world-wide communication infrastructure. Widely deployed and often misconfigured, embedded network devices constitute highly attractive targets for exploitation.

Although common wisdom enforces the suspicion that embedded devices tend to be less secure then general purpose computers and often trivial to exploit, evidence of such insecurities is mostly anecdotal. To fully appreciate the scope and scale of the embedded threat, we must move beyond analysis of individual embedded devices and their vulnerabilities. In order to formulate realistic and effective mitigation strategies against current and next generation embedded device exploitation, we first pose and answer several fundamental questions:

- How have embedded devices been exploited in the past? How feasible is large scale exploitation of embedded devices? (Section 2)
- How can we quantitatively measure the level of embedded device insecurity on a global scale? (Section 3)
- How can compromised embedded devices be used to benefit malicious attackers? (Section 4)
- How many vulnerable embedded devices are there in the world? What are they? Where are they? (Section 5)
- What are the most efficient methods of securing vulnerable embedded devices? (Section 6)

The purpose of our project is to quantify and trend the level of insecurity of embedded devices currently in the wild. To this end, we first establish an observed **lower bound** on the number of trivially vulnerable embedded devices on the internet. We do this by assuming the role of the least sophisticated malicious attacker (See Section 3.1), who only tries to log into publicly reachable embedded devices using well known **default root credentials**. Section 3 describes the default credential scanner we developed using standard tools such as **nmap**, which positively identified over **540,000** wide open embedded devices.

Vulnerable devices were detected in **144 countries**, in enterprise, ISP, government, educational, satellite provider as well as residential network environments[1]. We discov-

[1]Military networks are intentionally excluded from our scan, although a collaborative effort is currently underway to carry out the same scan on US military networks.

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

Total IPs Scanned	Devices Targeted	Vulnerable Devices	Vulnerability Rate
3,223,358,720	3,912,574	540,435	13.81%

Table 1: Scale and Result of the Latest Global Default Credential Scan.

ered vulnerable devices across a diverse spectrum of product types, including consumer appliances, home networking devices, office appliances, enterprise and carrier networking equipment, data-center power management devices, network security appliances, server lights-out-management controllers, IP camera surveillance systems, VoIP devices, video conferencing appliances as well as ISP issued modems and set-top boxes. Section 5 presents detailed analysis of the data collected by our default credential scanner.

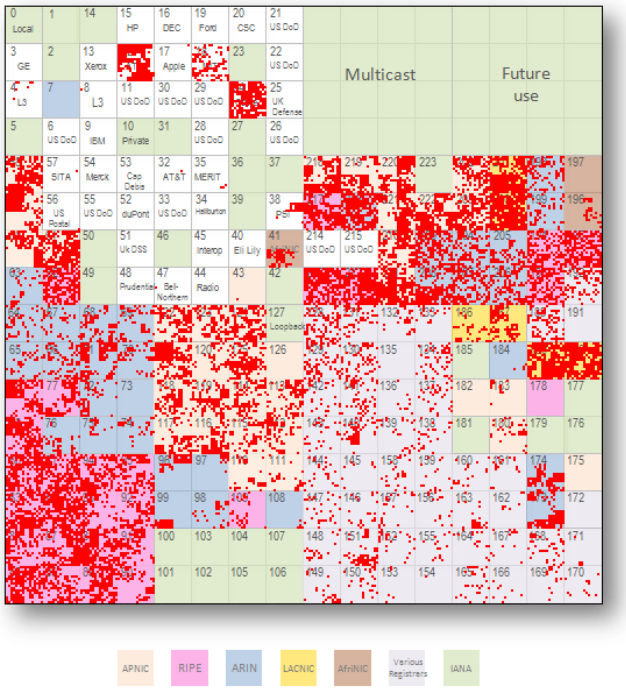


Figure 1: Distribution of Vulnerable Embedded Devices in IPv4 Space. Total Number of Vulnerable Devices Found: 540,435.

While the observed quantity and distribution of embedded devices configured with default root passwords demonstrate a global, pervasive phenomenon, we believe the data presented in this paper represent a conservative lower bound on the actual population of vulnerable devices in the wild. Evidence suggests that this lower bound can be raised significantly by slightly escalating the level of sophistication of our assumed attacker [11].

1.1 Contributions

We present the first quantitative measurement of embedded device insecurity on a global scale, along with preliminary results from an ongoing longitudinal study of the same subject. By assuming the role of the least sophisticated attacker (see Section 3.1), we present an observed **lower bound** on the distribution of trivially exploitable network embedded devices over functional (Section 5.1), spatial (Section 5.2), organizational (Section 5.3) and temporal (Section 5.5) domains.

The embedded device default credential scanner created for this experiment is designed to identify efficiently and safely the vulnerable embedded devices on the network. It does this by testing whether one can remotely login into a device using well known default root credentials. The verification process is designed to use minimal resources on the target embedded device. The scanner currently supports 73 unique embedded device types including consumer appliances, home networking devices, office appliances, enterprise and carrier networking equipment, data-center power management devices, network security appliances, server lights-out-management controllers, IP camera surveillance systems, VoIP devices, video conferencing appliances as well as ISP issued modems and set-top boxes.

While the embedded security threat has been generally known for some time, the data presented in this paper provides a real-world quantitative assessment of the scale and scope of the embedded threat on a global level. Analysis of our results yields several interesting features within the observed vulnerability distributions. The features presented in Section 5 presents insights into the root causes of the existence of vulnerable embedded devices. By combining the observed vulnerability distributions and its potential root causes, we formulate a set of mitigation strategies and hypothesize about its quantitative impact on reducing the global vulnerable device population.

Many forces will undoubtably change the observable lower bound of embedded device insecurity as time goes on. For example, the out-of-the-box security of new embedded products may change. Network operators controlling large homogeneous sets of devices may improve their security, as may small and medium size organizations like private enterprises and educational organizations. The level of malicious exploitation will also indirectly contribute to the overall effort dedicated to improving embedded device security. Lastly, it is our hope that the data and mitigation strategies reported in this paper will generate more awareness of this pervasive threat. In order to quantify the scope of the embedded device insecurity threat over time and detect such forces at work, we plan to continue our scanning activities to conduct an ongoing longitudinal study over the next year. Section 5.5 discusses the preliminary results of our longitudinal study over the past four months.

1.2 Outline

The remainder of this paper is organized as follows: Section 2 surveys recent developments related to embedded device insecurity in white-hat and black-hat communities as well as popular literature. Section 3 describes our methodology with emphasis on the steps taken to ensure a safe and ethical experimental protocol. Section 4 describes a variety of novel malicious uses of the vulnerable devices discovered by our scanner. Section 5 presents the analysis of data gathered from our latest global scan as well as preliminary results from our ongoing longitudinal study. Section 6 presents a set of remediation strategies, along with an quantitative estimates of its potential effect with respect to the global vulnerable device population. We conclude in Section 7 with a summary of our contributions.

2. RELATED WORKS

Evidence of embedded device insecurity and exploitation has been presented in both white-hat and black-hat venues for quite some time. The creation and propagation characteristics of hypothetical malnets exploiting vulnerable wireless routers have been described by several researchers [10, 19]. For example, Traynor *et al.* showed that an adversary can potentially compromise over 24,000 routers in Manhattan in less than 2 hours [19]. The data from our scan indicates that trivially exploitable embedded devices exist in sufficient quantity and concentration for such hypothetical attacks to be feasible. Our data also corroborates that phishing attacks using compromised consumer electronics such as home routers [20] can be carried out on a large scale by technically unsophisticated attackers.

Existing evidence clearly reenforces the common wisdom that embedded devices are generally less secure then general purpose computers and are often trivial to exploit. However, the available literature tends to focus on specific vulnerabilities or vulnerable devices.

For example, a recent Wired.com article [9] announced a vulnerability found on the administrative interface of the SMC8014 series cable modem, potentially affecting 65,000 Time Warner customers. Numerous research projects [18, 11] targeting specific device types have demonstrated that large numbers of vulnerabilities within ubiquitous embedded device types. According to Bojinov *et al.*, an audit of common embedded administrative interfaces from 16 major manufacturers yielded significant vulnerabilities from all of the 21 devices considered [11].

The evolution of embedded device exploitation tools and techniques demonstrate an accelerating maturation of malicious attacks against embedded devices. While proof of concept Cisco IOS exploits and shellcode have been publicly available since 2003 [13, 16], recent evidence suggests that attackers are scanning for and exploiting consumer routers to build modest size bot-nets, mainly for DDOS purposes. The appearance of tutorials [5] and simple to use tools to find and control specific consumer routers indicate that embedded device exploitation techniques are beginning to diffuse out of research circles, and into the general black-hat community.

To the best of our knowledge, the first consumer router botnet, psyb0t, was reported by Dronebl.org in 2008 [6]. While no detailed analysis of the bot was published, we do know that it primarily targeted mipsel OpenWRT and DD-WRT devices using default passwords. It is suspected that the psyb0t botnet observed in 2008 was a proof of concept test of the technology [7], as the botnet was quickly shutdown by its operators following Dronebl.org's public announcement of its existence.

The current generation of embedded device malcode may be related to existing unix tools like Kaiten.c [1]. A survey of black hat literature circa 2008 shows at least one document describing the process of compromising similar consumer routers using password guessing and existing unix IRC bots [5]. This may help to explain why the majority of victim embedded devices exploited thus far have been unix-based consumer routers. For example, psyb0t targeted only home routers and heavily leveraged the unix-like operating environment found on its victim devices. Specifically, psyb0t used commands like wget and chmod to download its payload onto victim devices and used iptables to block all administrative interfaces to protect the device from other attackers.

2.1 Next Generation Embedded Malcode

Existing embedded device malcode such as psyb0t depend heavily on its victim devices' similarity to traditional unix systems. While development of such malcode is relatively straightforward, it constrains the vulnerable population to low-end consumer appliances running unix-like operating systems. For example, enterprise networking devices like Cisco routers and switches run on proprietary operating systems like IOS, which do not resemble traditional unix architecture. However, recent advancements in exploitation and root-kitting techniques for proprietary operating systems like Cisco IOS [17, 14] could allow attackers to compromise high-end enterprise devices like backbone routers and firewalls. It is highly likely that the next generation of embedded device malcode will have greater ability to compromise heterogeneous device types, stealthier and more sophisticated command and control channels, as well as other malicious capabilities aside from DDOS.

Furthermore, as data presented in Section 5 suggest, the current population of trivially vulnerable embedded devices is quite high. Therefore, the next generation of malcode capable of compromising heterogeneous device types will easily be able to infect significantly more devices then psyb0t and kaiten.c in their current state.

3. EXPERIMENTAL METHODOLOGY

The default credential scanner is designed to quickly sweep large portions of the internet. Each scan takes approximately four weeks and involves two or three sweeps of the entire monitored IP space (Section 3.4 discusses how the monitored IP ranges are selected.)

Multiple sweeps across the same IP space is desirable for two reasons. First, embedded devices on residential networks have unpredictable availability. Therefore, multiple sweeps increase the scanner's probability of observing a vulnerable device when it is connected to the network. Second, multiple sweeps across the same address space over months and years allow us to conduct a **longitudinal** study on the vulnerability rates of embedded devices around the world.

In Section 5, we present the results of our latest scan, containing over 540,000 observed vulnerable devices, as well as analysis of preliminary data gathered by tracking approximately 102,000 vulnerable embedded devices over a span of four months in Section 5. This is an ongoing study, and we plan to publish the results of a detailed longitudinal study over the next year when the data becomes available.

3.1 Threat Model

For the sake of establishing a lower bound on the state of embedded device insecurity in the wild, we assume the role of the least sophisticated malicious attacker. The attacker has unrestricted access to the internet but is unable to exploit any vulnerabilities found on any devices. Instead, the attacker has access to the network scanner nmap and a list of well known factory default root credentials for popular network embedded devices.

For the remainder of the paper, we define a *vulnerable* device as any device that is reachable on the internet and allows the attacker to gain root privileges by using factory default credentials.

```
User Access Verification

Username:
```

Figure 2: Common Cisco Telnet Login Prompt.

```
root:
    username_prompt: ['sername:']
    username: cisco
    askuser: true
    passstr: ['assword:']
    incorrect: [sername, assword]
    success: ['\#']
    passwords: ['cisco']
    deviceType: cisco
    isActive: 'true'
```

```
root:
    authType: basicAuth
    passwds: ['admin']
    authRealm: WRT54G
    username: ''
    deviceType: linksys-wrt
    loginURL: '/'
    isActive: 'true'
```

3.2 Default Credential Scanner: A Three Phase Process

The default credential scan process is straightforward and can be broken down into three sequential phases: **recognizance**, **identification**, and **verification**.

Recognizance: First, nmap is used to scan large portions of the internet for open TCP ports 23 and 80. The results of scan is stored in a SQL database.

Identification: Next, the device identification process connects to all listening Telnet and HTTP servers to retrieve the initial output of these servers[2]. The server output is stored in a SQL database then matched against a list of signatures to identify the manufacturer and model of the device in question (See 3.3).
For example, Figure 2 shows a telnet login prompt common to Cisco routers and switches.

Verification: Once the manufacturer and model of the device are positively identified, the verification phase uses an automated script to attempt to log into devices found in the identification phase. This script uses only well known default root credentials for the specific device model and does not engage in any form of brute force password guessing. We create a unique *device verification profile* for each type of embedded device we monitor. This profile contains all information necessary for the verification script to automatically negotiate the authentication process, using either the device's Telnet or HTTP administrative interface. Figure 3.2 shows two typical device verification profiles, one for the administrative Telnet interface for Cisco switches and routers, the other for the HTTP administrative interface for Linksys WRT routers using HTTP Basic Authentication. Each device verification profile contains information like the username and password prompt signatures, default credentials as well as authentication success and failure conditions for the particular embedded device type. Once the success or failure of the default credential is verified, the TCP session is terminated and the results are written to an encrypted flash drive for off-line analysis. (See 3.5).

[2]In case of HTTP, we issue the 'get /' request

Total IPs Scanned	Number of Countries Scanned	Number of Organizations Scanned
3,223,358,720	193	17,427
Most Heavily Scanned Countries		
US	CN	JP
1,477,339,136	217,273,088	177,494,016
GB	DE	CN
111,457,280	107,387,648	77,328,896

Table 2: Key Statistics on the Scope and Geographical Distribution of the IP Ranges Currently Monitored by the Default Credential Scanner.

3.3 Device Selection

The full list of devices currently monitored by our default credential scanner can be found on our project webpage[3]. In order for an embedded device to be included in this list, its default root credentials must be well known and obtainable through either manufacturer documentation or simple search engine queries. The default credential scanner does not engage in any form of brute force password guessing.

The device selection process is manual and iterative. We begin by analyzing data gathered by the recognizance phase of our scanner, which collects the initial output from active Telnet and HTTP servers found by NMAP. We maintain three sets of signatures: non-embedded devices, non-candidate embedded devices and candidate embedded devices. Signatures of non-embedded devices include those of popular HTTP servers such as Apache and IIS as well as Telnet common authentication prompts of general purpose operating systems. Signatures of non-candidate embedded devices include those that do not ship with a well known default credential[4]. Signatures of candidate embedded devices include string patterns that positively identify the device as one that we are actively monitoring. After the recognizance data is tagged using these three signature sets, we manually inspect the remaining records, tagging, creating new signatures and device verification profiles.

3.4 Network Range Selection

We initially directed our scan towards the largest ISPs in North and South America, Europe and Asia. As we iteratively refined our scanning infrastructure, we gradually widened the scope of our scan to include select geographical locations within the United States. After testing our default credential scanner for over six months to ensure that it caused no harm to the scanned networks, we finally allowed the scanner to operate globally. Using a reverse lookup of the MaxMind GeoIP database [2], we included every /24 network in the IPv4 space which is associated to a geographical location. Table 2 shows some key metrics on the scope of the IP ranges which we currently monitor.

3.5 Ethical Considerations and Due Diligence

The technical methodology of our project is straightforward. However, the necessary means of gathering real-world data on the vulnerability rates of embedded device have raised an ethical debate.

[3]http://www.hacktory.cs.columbia.edu

[4]For example, the Polycom VSX 3000 video conferencing unit uses the device's serial number as the default password.

On one hand, the simple act of port scanning a remote network across the internet can be construed as a hostile and malicious attack. On the other hand, we can not move beyond vague and anecdotal suspicions of the embedded device security problem unless we gather large scale, quantitative evidence of the problem currently in the wild.

As advocated in a recent position paper on the ethics of security vulnerability research [15], this line of proactive vulnerability research serves an important social function and is **neither unethical nor illegal with respect to US law**.

The experimental results contain sensitive information on a large number of vulnerable devices in the world, some of which reside in sensitive environments. Therefore it is the responsibility of the research team to uphold a high standard for ethical behavior and due diligence when engaging in such sensitive research. The operating environment must be isolated and fortified against compromise and data exfiltration. Furthermore, each member of the research team must agree to adhere to a clear experimental protocol to ensure that **no harm is done**.

A trivial network scanner can be implemented with little work. However, using such a scanner openly on a global scale is irresponsible and ethically unacceptable. Therefore we have invested a large portion of of energy to create a secure research environment and a responsible experimental protocol in order to ensure that our activities cause no harm:

Doing no harm. Bound by the ethics principal of the duty not to harm, we have taken numerous steps to ensure that our research activities do not interfere with the normal operations of the networks we monitor. To this end, the default credential scanner is designed to use minimal external resources in order to accurately verify device vulnerability. We scan target networks in /24 blocks in non-sequential order in order to minimize the number of incoming TCP requests destined to any individual organization. Detailed activity logs are kept to ensure that no device or network is unnecessarily probed multiple times during a single scan. Overall, non-embedded devices and non-candidate embedded devices will receive at most 6 TCP packets over a period of several minutes. The scanner's outbound packet-rate is policed and monitored in order not to overwhelm any in-path networking devices. Lastly, each IP address used by our scanner runs a pubic webpage describing the intention and methodology of our project [3]. This page also provides instructions for permanently opting-out of the scan. (See Table 6). Such requests are monitored by both our research team as well as the Columbia University NOC, and are promptly honored without question.

Implementing a secure research environment. The scan system is contained in a DMZ network behind a Cisco ASA firewall. Scanning nodes are isolated from the university network. Inbound access to this protected network can only be established by using IPSec VPN. Outbound access by the scanning nodes are limited to the ports which they are scanning (Telnet, HTTP, etc).

Compartmentalization of access to sensitive information. VPN access to the scan system DMZ is granted only to active members of the research team. New students participating in research are first given access to a separate DMZ containing a development copy of the scan system with no sensitive data. Access to the production environment is given to students only after they have acknowledged and demonstrated understanding of the experimental protocol.

Proper handling of sensitive data at rest. Sensitive experimental data is purged from the production database regularly, then transferred to an IronKey [4] USB stick for encrypted offline storage. This is done to minimize the amount of data available for exfiltration in case of a compromise of the research environment.

Notifications of vulnerabilities through trusted channels. Significant vulnerabilities are reported to Team Cymru, who brokers communications between our research team and the appropriate contacts. Sensitive information detailing the vulnerable devices is either physically handed off to Team Cmyru members or transferred using encrypted channels.

4. MALICIOUS POTENTIAL OF EMBEDDED DEVICE EXPLOITATION

This section discusses several novel ways of exploiting vulnerable embedded devices due to their unique functions and hardware capabilities. After auditing the functional capabilities of many different embedded devices, we have concluded that the attacks described below are trivially possible among a majority of embedded devices within the appropriate functional categories. All attacks discussed below can be carried out through legitimate manipulation of the administrative interface. More importantly, as the data presented in Section 5 illustrate quantitatively, there exists a large population of embedded devices vulnerable to each of the attacks discussed below. Although DDOS attacks using embedded devices have certainly been carried out on a relatively large scale, most of the other attacks described in this section have not. However, considering the data presented in Section 5, we posit that it is only a matter of time before such attacks are carried out systematically on a large scale.

We have engaged several major organizations to mitigate some of the issues discussed below. Therefore, specific details regarding organization names and device model information are withheld when appropriate.

4.1 Massive DDOS Potential

The heterogeneous nature of embedded administrative interfaces makes orchestrating large DDOS attacks using embedded devices a logistic challenge. Vulnerable embedded devices clearly exist in large numbers in the wild. However, it is often believed that embedded operating systems are too diverse; and capturing the long tail of this diversity is required to carry out large scale exploitation. Data gathered by our default credential scanner reveal that many large vulnerable homogenous device groups exist in the wild. In fact, the top **3** most vulnerable device types represent over **55%** of all vulnerable devices discovered by our latest scan. In other words, there exists at least 300,000 vulnerable embedded devices which can be controlled via 3 similar Telnet-based administrative interfaces. The exact model of these three device groups have been anonymized. However, these three device groups are centrally managed by various service providers around the world, and thus can be systematically

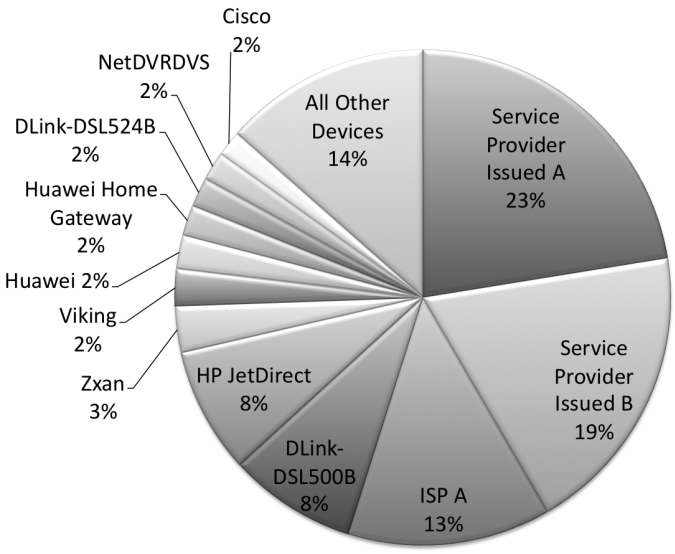


Figure 3: Distribution of Vulnerable Devices Across Unique Device Types. The Top 3 Device Types Constitute 55% of the Entire Vulnerable Device Population.

secured in a feasible manner. Figure 3 shows the distribution of the top 12 most frequently encountered vulnerable embedded device types.

4.2 VoIP Appliance Exploitation

VoIP adapters like the Linksys PAP2, Linksys SPA and Sipura SPA are consumer appliances, which provide a gateway between standard analog telephones and VoIP service providers. In many cases, the publicly accessible HTTP interface of such devices will display diagnostic information without requiring any user authentication. This information usually includes the name of the customer, their phone number(s), a log of incoming and outgoing calls, and relevant information regarding the SIP gateway to which the device is configured to connect. Once authenticated as the administrative user, an attacker can usually retrieve the customer's SIP credentials, either by exploiting trivial HTTP vulnerabilities[5] or redirecting the victim to a malicious SIP server.

4.3 Data Leakage via Office Appliance Exploitation

Enterprise printers servers and digital document stations are ubiquitous in most work environments. According to our data, network printers also constitute one of the most vulnerable types of embedded devices. For example, our default credential scanner identified over **44,000** vulnerable HP JetDirect Print Servers in **2,505** unique organizations worldwide. Since high-end print servers and document stations often have the capability of digitally caching the documents it processes, we posit that an attacker can use such devices not only to monitor the flow of internal documents, but also to exfiltrate them as well.

[5]Credentials are sometimes displayed in clear-text within HTML password fields. While this appears to hide the passwords in the web browser, it does not hide it in the HTML source.

4.4 Enterprise Credential Leakage via Accidental Misconfiguration

It is common practice for organizations that operate large homogenous collections of networking equipment to apply the same set of administrative credentials to all managed devices. While this significantly reduces the complexity and cost of managing a large network, it also puts the network at risk of total compromise. Using a single master root password for all networking devices is safe so long as every device is correctly configured at all times, and the master password is not leaked. If an enterprise networking device is brought online with both factory default credentials, as well as the master credentials of the organization, an attacker can easily obtain the master root password for the entire network. While this event is unlikely, the probability of such a misconfiguration quickly increases with the size and complexity of the organization, specially when human error is taken into account. We have not verified that such an attack is feasible; however, our data indicate that enterprise networking devices residing within large homogenous environments have been misconfigured with default root credentials.

5. ANALYSIS OF RESULTS

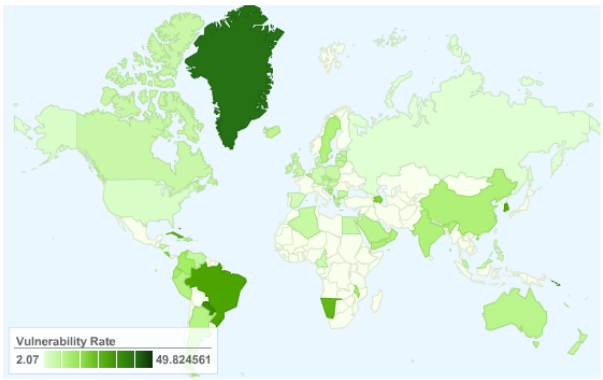


Figure 4: Embedded Device Vulnerability Rates of Monitored Countries (Threshold = 2%).

In this section we present latest data gathered by our default credential scanner as well as preliminary results from our ongoing longitudinal study, tracking approximately 102,000 vulnerable devices over a span of four months. We also present statistics on the level of human and organizational responses received by Columbia University regarding our scanning activities. Figure 4 shows a heat map of embedded device vulnerability rates across monitored countries.

Section 5.1 shows the breakdown of vulnerable embedded devices across **9 functional categories**; Enterprise Devices, VoIP Devices, Home Networking Devices, Camera/Surveillance, Office Appliances, Power Management Controllers, Service Provider Issued Equipment, Video Conferencing Units, and Home Brew Devices. Section 5.2 shows the breakdown of vulnerable embedded devices across **6 continents**. Section 5.3 shows the breakdown of vulnerable devices across **5 types of organizations**; Educational, ISP, Private Enterprise, Government, and Unidentified.

5.1 Breakdown of Vulnerable Devices by Functional Categories

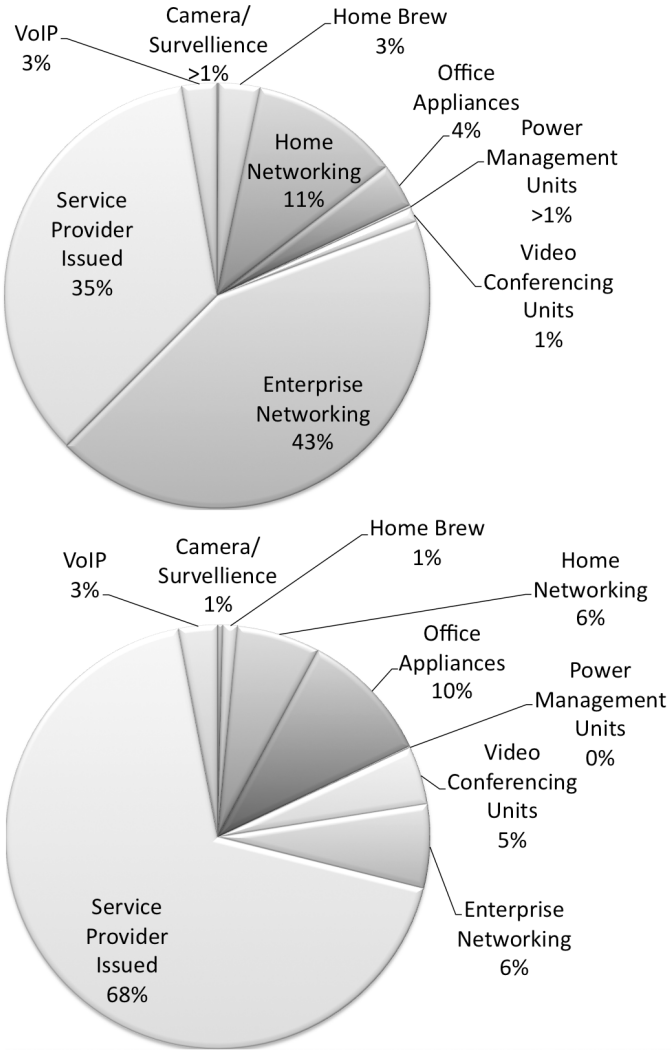


Figure 5: Discovered Candidate Devices (Top) and Vulnerable Devices (Bottom) By By Organization Type.

We organized 73 unique embedded device types monitored by our scan into 9 functional categories. Detailed categorization of monitored devices can be found on our project webpage[6]. Figure 5 shows the distribution of all discovered candidate embedded devices (top) and the distribution of vulnerable embedded devices (bottom) across the different functional categories. Table 3 shows the total number candidate embedded devices discovered within each functional category as well as their corresponding vulnerability rate.

- While **Service Provider Issued Equipment** accounts for only 35% of all discovered candidate embedded devices, it represents 68% of all vulnerable embedded devices.
- While **Enterprise Networking Equipment** accounts for 43% of all discovered candidate embedded devices, it only represents 6% of all vulnerable embedded devices.

[6]http://www.hacktory.cs.columbia.edu

5.2 Breakdown of Vulnerable Devices by Geographical Location

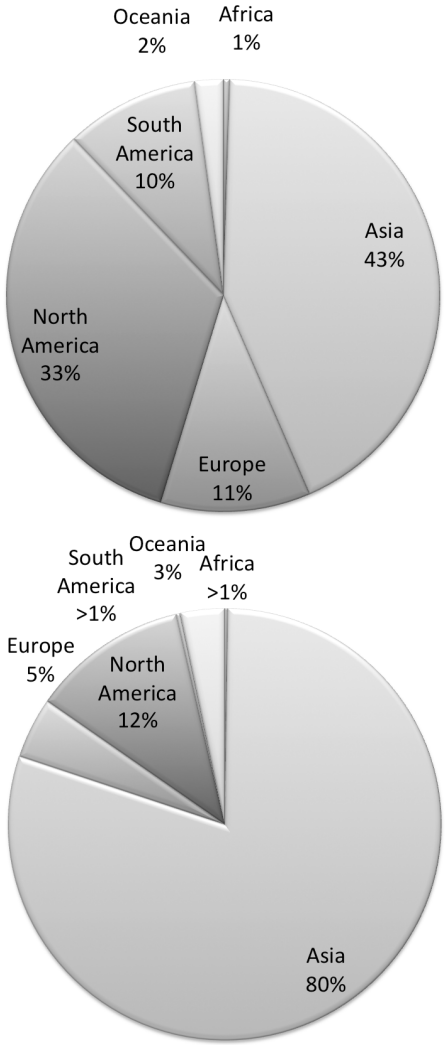


Figure 6: Discovered Candidate Devices (Top) and Vulnerable Devices (Bottom) By Geographical Distribution.

Using the MaxMind GeoIP database[2], we categorized all discovered candidate and vulnerable embedded devices according to the continent in which they are located. Figure 6 shows the distribution of all discovered embedded devices (top) and the distribution of vulnerable embedded devices (bottom) across 6 continents. Table 4 shows the total number of candidate embedded devices as well as the corresponding vulnerability rate within each continent.

- **Asia** represents the continent with the most number of candidate embedded devices and accounts for approximately 80% of all discovered vulnerable embedded devices.
- **South Korea** contains the largest number vulnerable embedded devices out of all monitored nations.
- While 33% of all discovered candidate embedded devices reside within **North America**, only 12% of all vulnerable embedded devices are found there.

5.3 Breakdown of Vulnerable Devices by Organizational Categories

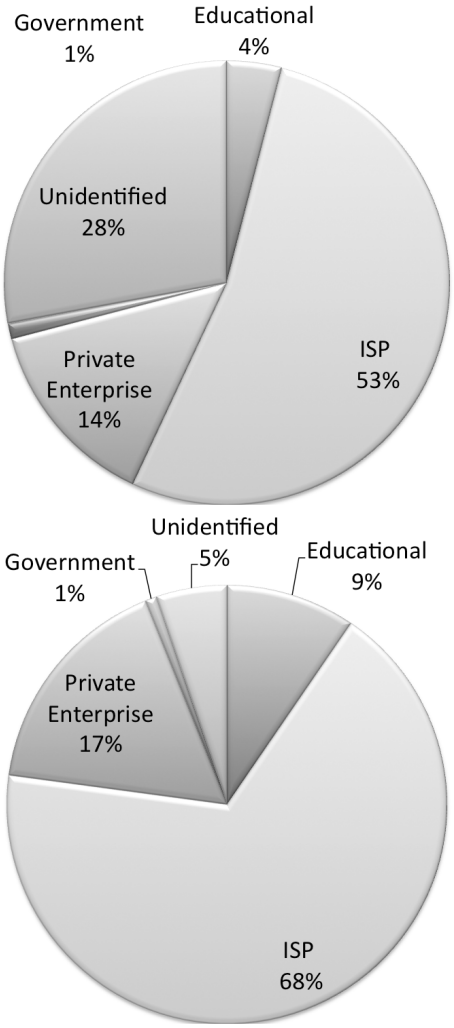


Figure 7: Discovered Candidate Devices (Top) and Vulnerable Devices (Bottom) By By Organization Type.

Using the MaxMind GeoIP Organization database[2], we categorized all monitored network ranges into 17,427 individual organizations. This was then divided into 4 general organization types; Educational, Internet Service Provider (ISP), Private Enterprise, and Government. 9118 organizations could not be accurately classified, and were left in Unidentified category. Figure 7 shows the distribution of all discovered embedded devices (top) and the distribution of vulnerable embedded devices (bottom) across the 5 organization types. Table 5 shows the total number of candidate embedded devices as well as the corresponding vulnerability rate within each organization type.

- **ISP** networks contain the most number of candidate embedded devices and house over 68% of all discovered vulnerable embedded devices.
- While **Educational** networks contain only a modest number of candidate embedded devices, it has the highest per category vulnerability rate of 32.83%

	Vul. Rate	Total Devices
Enterprise Devices	2.03%	1,689,245
VoIP Devices	15.34%	104,827
Home Networking	7.70%	445,147
Camera/Surveillance	39.72%	5,080
Office Appliances	41.19%	132,991
Power Management	7.23%	7,429
Service Provider Issued	27.02%	1,362,347
Video Conferencing	55.44%	43,349
Home Brew	4.93%	122,159

Table 3: Vulnerability Rate by Device Category.

	Vul. Rate	Total Devices
Africa	5.36%	19,363
Asia	21.69%	1,731,089
Europe	4.76%	450,019
North America	4.12%	1,335,575
South America	0.37%	402,163
Oceania	17.98%	85,941

Table 4: Total Discovered Candidate Embedded Devices and Corresponding Vulnerability Rates By Geographical Location (Continental).

	Unique Orgs	Vul. Rate	Total Devices
Educational	1,371	32.83%	156,992
ISP	2,374	17.43%	2,095,292
Priv. Enterprise	4,070	16.40%	554,101
Government	494	10.38%	44,460
Unidentified	9,118	2.54%	1,103,775

Table 5: Vulnerability Rate By Organization Type.

5.4 Community Response to Default Credential Scanner Activity

The default credential scanner is designed to direct interested parties to a public webpage which describes the intent and methodology of our project[3]. Each IP address used by the scanner also hosts a public HTTP server which redirects visitors to the public project webpage. We tracked access to this webpage using Google Analytics as a way to gauge the global community's awareness of our scanning activities. Figure 8 shows the number and geographical distribution of visitors over the past six months. The initial spike of visitors in October 2009 coincided with the publication of an article regarding preliminary results of our project [8]. Since then, our continuous scanning activity attracted **87 visitors** over the last 5 months.

Total Conversations	Opt-Out Requests	Request for Information, but Not Opt-Out
36	14	22
Tone of Counter-Party		
Supportive	Neutral	Hostile
14	15	7

Table 6: Email Correspondences Received from Network Operators Regarding Scanning Activity.

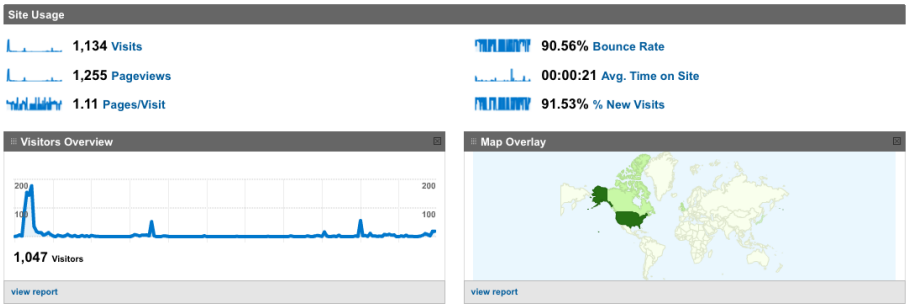


Figure 8: Daily Page Access Analytics For Scan Project Information Page [www.hacktory.cs.columbia.edu]. Oct 19, 2009 - April 12, 2010.

Vulnerable Devices Tracked	102,896
Tracked Devices Currently Online	54,429
Tracked Devices Currently Vulnerable	52,661

Table 7: Preliminary Longitudinal Study Tracking 102,896 Vulnerable Devices Over 4 Months.

Table 6 shows a breakdown of all communications between the operators of the networks monitored by our scanner and our research team. The conversations were all initiated by the counter-party via email, usually requesting further information or to be excluded from the scan. We answered 36 conversations in total, 14 of which requested certain IP ranges to be permanently excluded. 1,798 /24 networks were excluded as a result of these requests. **61%** of all interested parties which detected our scanning activity and contacted us decided to allow the scan to continue. The geographical location of the counter-parties correlates closely to the heat map in Figure 8. We did not receive any correspondence from ISP organizations or organizations from Asia, even though the majority of vulnerable devices were discovered within such IP ranges.

5.5 Preliminary Longitudinal Results

Table 7 shows the preliminary results of our longitudinal study. We retested 102,896 vulnerable embedded devices discovered at the end of December, 2009. As of April 20, 2010, 54,429 of the retested devices are still publicly accessible, out of which 52,661 devices remain vulnerable.

In other words, approximately **96.75%** of accessible vulnerable devices are still vulnerable after a 4 month period, and factory default credentials have been removed from only **3.25%** of the same set of devices.

6. REMEDIATION STRATEGY

The least sophisticated attacker modeled in this experiment can be defeated by simply discontinuing the use of well-known default credentials on embedded devices. However, the overall cost of implementing this naive mitigation strategy will likely be quite high in reality. In the unlikely event that all embedded device manufacturers universally agree to discontinue the use of well-known default passwords henceforth, we are still faced with the challenge of retroactively fixing the vulnerable legacy embedded devices in use throughout the world today. Therefore, it is reasonable to assume that the embedded security threat will likely persist and grow endemically for the near future. In order to effectively reduce the total population of vulnerable embedded devices in the wild, we must carefully consider the best methods for securing existing legacy devices. Since existing devices are by definition under the administrative control of some individual or organization, successful mitigation strategies must actively engage these network operators in order to fix the problem.

According to the data presented in Section 5, a few groups of network operators contribute disproportionally large numbers of vulnerable embedded devices to the global population. For example, we discovered over 300,000 vulnerable embedded devices operating in homogenous environments within two ISP networks in Asia. Overall, embedded devices operated by residential ISPs constitute over 68% of the entire vulnerable population. Since ISPs centrally manage large numbers of vulnerable embedded devices, they are the ideal candidates to engage to mitigate the embedded security threat.

While immediately effective, engaging individual organizations and manufacturers to fix pockets of vulnerable devices can only impede the growth of the embedded security threat but not solve it. In order to improve categorically the security posture of both new and legacy embedded devices, we must develop methods of delivering effective host-based protection onto large numbers of proprietary embedded devices running heterogeneous operating systems. We believe that a novel, injectable code structure called Parasitic Embedded Machines (PEM) [12] currently under development by the Columbia Intrusion Detection Systems Lab provides a viable solution to this challenging problem.

7. CONCLUSION AND FUTURE WORKS

We presented the first quantitative measurement of embedded device insecurity on a global scale as well as a preliminary longitudinal study tracking vulnerable embedded devices over a 4 month period. We developed an embedded device default credential scanner capable of efficiently and safely identifying vulnerable embedded devices on the network. The scanner does this by testing whether one can remotely login into a device using its well-known manufacturer supplied default credentials. Using this scanner, which currently monitors 73 common embedded device types, we identify over **540,000** publicly accessible vulnerable devices in 144 countries. Vulnerable embedded devices were discovered in 17,427 unique organizations on 6 continents including government, ISP, private enterprise, educational and satel-

lite provider networks. Preliminary results from our longitudinal study tracked 102,896 vulnerable devices discovered in December 2009. Out of the 54,429 devices currently online from the original population, **96.75%** such devices still remain vulnerable today. By breaking down the observed vulnerable embedded device population across functional, geographical and organizational categories, we were able to identify key groups which contribute a disproportionately large number of vulnerable devices to the global population. Lastly, using observations derived from the presented data, we proposed a set of realistic mitigation strategies to effectively reduce the total population of vulnerable embedded devices. This study demonstrates that there is a very large population of trivially vulnerable embedded devices available for exploitation by the least sophisticated adversary. We posit that the size of this vulnerable population can be significantly increased by escalating the level of sophistication of the assumed attacker. Since no widely available host-based defenses exist, vulnerable embedded devices constitute a serious and pervasive security problem.

8. REFERENCES

[1] kaiten.c IRC DDOS Bot. http://packetstormsecurity.nl/irc/kaiten.c.

[2] MaxMind GeoIP. http://www.maxmind.com/app/ip-location.

[3] Embedded Device Vulnerability Assessment Initiative. http://www.hacktory.cs.columbia.edu.

[4] IronKey Personal D200. http://www.ironkey.com/personal-solutions.

[5] The End of Your Internet: Malware for Home Routers, 2008. http://data.nicenamecrew.com/papers/malwareforrouters/paper.txt.

[6] Network Bluepill. Dronebl.org, 2008. http://www.dronebl.org/blog/8.

[7] Psyb0t' worm infects linksys, netgear home routers, modems. ZDNET, 2009. http://blogs.zdnet.com/BTL/?p=15197.

[8] Scan of internet uncovers thousands of vulnerable embedded devices. http://www.wired.com/threatlevel/2009/10/vulnerable-devices/, 2009.

[9] Time warner cable exposes 65,000 customer routers to remote hacks. http://www.wired.com/threatlevel/2009/10/time-warner-cable/, 2009.

[10] P. Akritidis, W. Y. Chin, V. T. Lam, S. Sidiroglou, and K. G. Anagnostakis. Proximity breeds danger: Emerging threats in metro-area wireless networks. In *In Proceedings of the 16 th USENIX Security Symposium*, pages 323–338, 2007.

[11] Hristo Bojinov, Elie Bursztein, Eric Lovett, and Dan Boneh. Embedded management interfaces: Emerging massive insecurity. Black Hat USA, 2009, 2009.

[12] Ang Cui and Salvatore J. Stolfo. Generic rootkit detection for embedded devices using parasitic embedded machines. Columbia University, New York. cucs-009-10., 2010.

[13] Felix "FX" Linder. Cisco Vulnerabilities. In *In BlackHat USA*, 2003.

[14] Felix "FX" Linder. Cisco IOS Router Exploitation. In *In BlackHat USA*, 2009.

[15] Andrea M. Matwyshyn, Angelos D. Keromytis Ang Cui, and Salvatore J. Stolfo. Ethics in security vulnerability research. *IEEE Security and Privacy (Vol. 8, No. 2)*, 2010.

[16] Michael Lynn. Cisco IOS Shellcode, 2005. In BlackHat USA.

[17] Sebastian Muniz. Killing the myth of Cisco IOS rootkits: DIK, 2008. In EUSecWest.

[18] Petko D. Petkov. Router Hacking Challenge, 2008. http://www.gnucitizen.org/blog/router-hacking-challenge/.

[19] Patrick Traynor, Kevin R. B. Butler, William Enck, Patrick McDaniel, and Kevin Borders. malnets: large-scale malicious networks *ia* compromised wireless access points. *Security and Communication Networks*, 3(2-3):102–113, 2010.

[20] Alex Tsow. Phishing with consumer electronics - malicious home routers. In Tim Finin, Lalana Kagal, and Daniel Olmedilla, editors, *MTW*, volume 190 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.

Multi-vendor Penetration Testing in the Advanced Metering Infrastructure

Stephen McLaughlin, Dmitry Podkuiko, Sergei Miadzvezhanka,
Adam Delozier, and Patrick McDaniel
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA 16802
{smclaugh,podkuiko,swm5344,delozier,mcdaniel}@cse.psu.edu

Abstract - The advanced metering infrastructure (AMI) is revolutionizing electrical grids. Intelligent AMI "smart meters" report real time usage data that enables efficient energy generation and use. However, aggressive deployments are outpacing security efforts: new devices from a dizzying array of vendors are being introduced into grids with little or no understanding of the security problems they represent. In this paper we develop an *archetypal attack tree* approach to guide penetration testing across multiple-vendor implementations of a technology class. In this, we graft archetypal attack trees modeling broad adversary goals and attack vectors to vendor-specific concrete attack trees. Evaluators then use the grafted trees as a roadmap to penetration testing. We apply this approach within AMI to model attacker goals such as energy fraud and denial of service. Our experiments with multiple vendors generate real attack scenarios using vulnerabilities identified during directed penetration testing, e.g., manipulation of energy usage data, spoofing meters, and extracting sensitive data from internal registers. More broadly, we show how we can reuse efforts in penetration testing to efficiently evaluate the increasingly large body of AMI technologies being deployed in the field.

1. INTRODUCTION

The Advanced Metering Infrastructure (AMI) is changing the way electric energy is produced, priced, and consumed. The introduction of digital sensors–*smart meters*–in homes and enterprises has allowed regional and national producers to more efficiently produce and deliver energy [18]. In short, the vast yet antiquated analog control system that has served electricity consumers for decades is entering the information age. Here AMI is evolving and being deployed quickly. In the US, the recent stimulus package allocates US $4.5 billion for smart grid technology development [25], with the energy sector making substantial additional investments. Similar efforts are under way internationally, with the EU, Canada, and China launching broad initiatives in recent years. Such expenditures are driving the dizzying array of new products that reach the market almost every day.

The transition of electric meters to digital systems is not without risks. New technologies offer new opportunities for adversaries to manipulate the grid to further their malicious ends. Moreover, deployments are outpacing security efforts: new devices and technologies are being introduced into grids with little or no real understanding of the security problems they represent. Current penetration testing efforts are piecemeal, ad hoc and often superficial. Not surprisingly, new vulnerabilities are being found almost as quickly as AMI products are being deployed [33, 20, 9].

Prudence critically demands better analyses of AMI system security: manufacturers and utilities must leverage modeling and analysis efforts for the large body of systems towards a global understanding of the security problems they represent. Efforts like the NIST smart grid guidelines [30] are a step in the right direction, but only identify affirmative steps for secure systems. They do not posit the causes and effects of critical vulnerabilities, nor identify a roadmap for offensive testing of smart meter technology. In the absence of guidance on these key issues, current industrial pen-testing strategies focus on specific vendor lines and are agnostic to critical security concerns–such as utilities' concerns with revenue protection from fraud and cost of operations.

In this paper, we design and execute a systematic penetration testing process for AMI systems and uncover a number of real attacks on commercially available systems. Our contributions in this effort include:

- We develop a new approach to guiding penetration testing. This approach uses vendor independent *archetypal attack trees* to model broad adversary goals and attack vectors, and *concrete attack trees* to instantiate specific attack subgoals on vendor systems.
- We develop archetypal and concrete attack trees for three important classes of attacks, (a) energy fraud, (b) denial of service, and (c) targeted disconnect. These trees represent practical (and in some cases trivial) attacks that can be carried out in widely deployed AMI systems.
- We identify from our penentration testing results of one and one half years a broad range of security vulnerabilities for two popular AMI vendors, and use them to instantiate real attack scenarios in fielded systems.

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

Representative attacks include the manipulation of energy usage data and signaling as it traverses public networks, spoofing meter identity, and physically extracting sensitive data from meters.

In this work, we focus solely on AMI: neighborhood-level smart grids including smart meters, utility management services, and the communications between them. However, there is nothing specific to AMI in the archetypal attack tree approach. The explored techniques are applicable to a broad range of products such as SCADA, medical devices, or automotive systems. We begin the exploration of this approach and its use in the next section.

2. METHODOLOGY

An attack tree is a structure for enumerating the kinds of attacks that achieve a particular adversarial goal [27]. It does this by recursively breaking down a goal into finer- and finer-grained subgoals and finally to a set of attacks that achieve the original goal. An example attack tree that formed the genesis of this work [24] is shown in Figure 1. The root specifies the end goal, committing energy fraud by forging the energy usage information reported to the utility. The internal nodes (those with parents and children) describe the different combinations of conditions that must be met to commit fraud. Finally, the leaves of the tree are the attacks necessary for energy fraud. The final attribute of the tree is the conjunctions (AND/OR) between each layer of child nodes. These specify whether all or just one of the child branches must be followed to reach the goal in the parent node.

What we notice about this example is that the attacks at the leaves of the tree are fairly general, and seem applicable to most smart metering systems. This suggests that this type of tree is a widely applicable tool. However, because it lacks details about any specific system, its usefulness is limited in finding concrete vulnerabilities. Because we are pen-testing multiple commercially available metering systems, we will want to further specify the details of each attack in this generic tree. Thus, as we learn about the individual systems, we extend this generic tree with vendor-specific attack strategies. These ideas can be refined into two types of attack trees: *archetypal* and *concrete*.

The process of grafting a concrete tree to an archetypal tree is shown in Figure 2. For a given adversarial goal, one may define an *archetypal tree* that enumerates strategies for reaching the goal against any system of a given architecture. In the case of the example above, the goal is forged energy demand and the architecture is smart metering. Each leaf of an archetypal tree is an *archetypal attack*. A concrete tree then refines an archetypal attack with respect to a specific vendor's system. The subgoals in the concrete tree sensitive to the security mechanisms present in the system, and thus define the exact conditions under which the root goal can be achieved. The leaves of the concrete tree are the *concrete attacks* which ultimately allow an adversarial goal to be achieved. For the purposes of our study, we use penetration testing to determine the feasibility of each concrete attack.

Our method is similar to that originally used for attack patterns [14, 10]. An attack pattern is a parameterized description of an attack, e.g. an injection attack, that is generic until its parameters are instantiated. Attack patterns may be described in terms of attack trees. When considering a particular attack against a particular *instance* of a system, e.g. a company's network, its parameters are instantiated with the specific details of that system. The concept of attack trees is based on that of fault trees, which were originally use to model the dependencies between potential faults in aviation and nuclear power systems [8, 32].

Attack trees by themselves are useful as a guide for penetration testing. However, once the knowledge of system interfaces has been exhausted and the concrete attacks are developed, we resort to standard pen-testing techniques such as reverse engineering [6], fuzz testing [28], and the construction of custom attack tools. For example, we later examine an energy fraud attack based on a meter spoof program written in Python.

Documented throughout, our methodology for directing penetration testing includes:

1. **Capture architectural description:** Elicit the features of a general architecture for target domain (see Section 3).
2. **Construct archetypal tree:** Given the architectural description, design a generic and comprehensive archetypal tree for each adversarial goal (see Section 4).
3. **Capture vendor-specific description:** Identify the structures and security mechanisms present the Systems Under Test (SUTs) that may thwart a given archetypal attack (see Section 5).
4. **Construct concrete trees:** Graft the vendor-specific goals to an archetypal goal to form concrete trees (see Section 6).
5. **Perform Penetration Testing:** Attempt to achieve the concrete goals by performing penetration testing on the SUT (see Section 7).

3. THE ADVANCED METERING INFRASTRUCTURE

AMI may be divided into utility-side management, smart electric meter deployments, and the networks that connect these two. This section describes these three along with AMI security concerns. At the edge of the AMI resides its main component: smart electric meters. A smart meter is a digital equivalent of a stand-alone electromechanical meter. Their most distinguishing characteristic is the use of two-way network communication with utilities. Smart meters have evolved from early Automated Meter Reading (AMR) systems [5] to allow for automatic updates of dynamic pricing information [26] and curtailment of individual loads when the grid is under stress [11]. Internal storage is used to keep time of day measurements for Time of Use (TOU) pricing schemes [18] and logs for both power outages [17] and potential intrusions, the latter of which is further explored in section 5.1.

3.1 Smart Meter Architectures

A smart meter is a networked embedded system equipped with a special apparatus for sensing electrical currents flowing through wires. In this section, we tease out the details of this definition, starting with the individual computing platform and finishing with the network. Unless otherwise specified, the features described in this section are present in the vast majority of commercial smart meters.

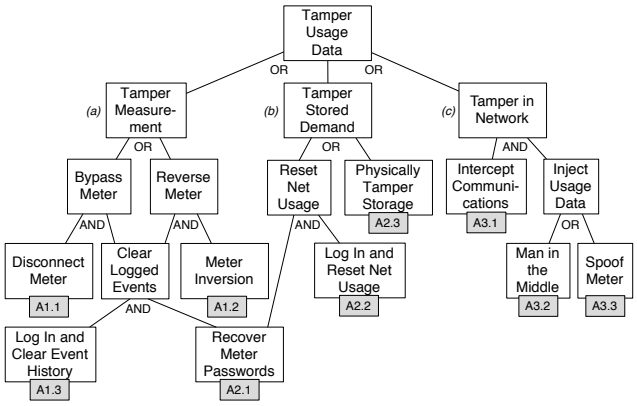


Figure 1: Example energy fraud attack tree. The three subgoals beneath the root are labeled as (a), (b), and (c) for reference purposes.

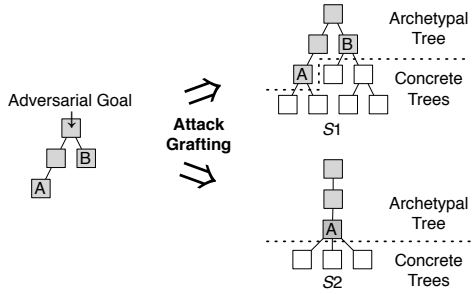


Figure 2: Grafting concrete trees for two different systems ($S1$ and $S2$) onto an archetypal attack tree for a specific adversarial goal.

Meters that are kept outside, such as those in the US, reside in protective socket enclosures, while those kept inside, which is common in the EU, often do not require a socket. The meter's internals are further protected by its cylindrical housing which consists of a base and a removable cover. To detect tampering by removal of the cover, a "flag" style aluminum tamper seal connects the cover to the base. This inexpensive seal consists of a stem which must be broken to remove the cover and a flag with a stamped identifier for the seal. As one might expect, there are no restrictions preventing the purchase of the seal with whatever flag marking is desired, making the removal of a seal for the purposes of physical tampering inconsequential.

The activities of a smart meter are coordinated by its Microcontroller Unit (MCU). The majority of work done by the MCU involves retrieving energy measurements from the low-level *meter engine* and storing them in flash memory for later transmission to the utility. Smart meter storage, however, is not used for electrical measurements alone. Like any general-purpose system, smart meters maintain logs of event histories and operating conditions. While the set of logged events varies between meter vendors, we cover the logs relevant to our security analysis later.

For flexibility of installation, smart meters within the same deployment can communicate over a number of different network mediums and topologies. Thus, meter firmware is designed to support a generic communication interface, leaving the specifics of a given network to a pluggable Network Interface Card (NIC). The meter exports a generic serial interface to communicate with the NIC, leaving the processing of specific network communication to the NIC.

If a meter is out of network communication with the utility and configurations or repairs are needed, it can be controlled locally through a standard infrared optical port located on its front panel. These ports are accessed via a small optical probe consisting of an LED and a photo-sensor at the range of less than one inch. While most meter vendors follow the physical layer standard for this port [4], the application layer is often proprietary. Typically, the optical ports transmit all data in the clear including passwords for user authentication. This includes the meter's administrator password.

One final component that deserves attention is the remote disconnect switch. If a utility wishes to disconnect a customer's power, it may do so remotely by transmitting a request to the meter to open the switch. The request is received by the digital portion of the meter, which issues the signal to the switch to break the circuit for the power flowing through the meter.

3.2 Meter Networks and Utility-Side Management

Given the sheer size of a utility's customer base, achieving networking connectivity with a meter at each individual home is a serious logistical challenge. Given the near impossibility of placing each individual meter on a public network, smart meters are designed to form their own LANs, each of which relies on a gateway device for communication between the LAN and public network. Some common choices of LAN and public network configurations are shown in Figure 3.

In the most common meter LANs, meters are connected in an adaptive wireless mesh network. Each meter in the mesh is a *repeater* that propagates data through the LAN to a *collector*. In some cases, the collector may itself be a meter. Power Line Communication (PLC) networks piggyback signalling over power distribution lines to form a star network topology that directly connects each meter in the LAN with the collector. The collector connects to the utility via a *backhaul* network such as the cellular or landline phone network, or the Internet.

On the utility end of the meter network resides a PC or server machine responsible for performing all regularly scheduled interactions with the meter. This machine runs a commodity OS, e.g. Microsoft Windows, a database server and the proprietary meter server software. If the utility server is compromised, the entire meter deployment is compromised.

3.3 AMI Security Concerns

Since smart meters have first come under scrutiny, concerns have been raised regarding their accuracy, reliability, security and privacy [23]. Academic and industrial pentesting efforts have found flaws in meter hardware [20], firmware [9] and network protocols [24]. Recently, Pacific Gas and Electric (PG&E) has experienced problems with measurement accuracy and meter network connectivity in their 5 million meter deployment, one of the largest in the US [15]. The

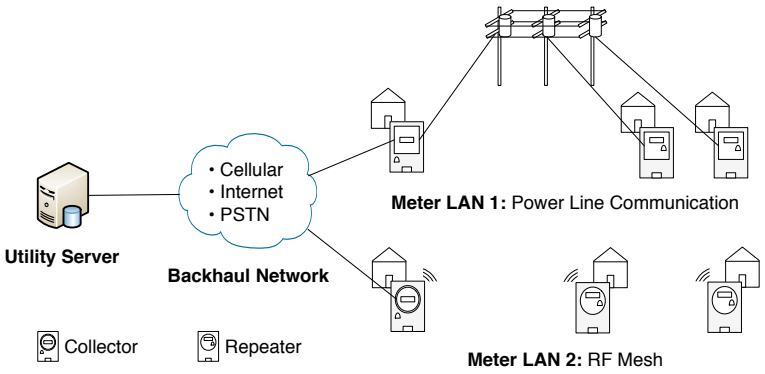


Figure 3: Connectivity of meters to utilities given two configurations of meter LANs.

addition of networks of such large numbers of devices to the uncontrolled Internet has been known to leave systems vulnerable to Denial of Service (DoS) attacks stemming from incompatibilities between their rigid proprietary designs and the Internet's open architecture [7, 31]. It will later be shown that this is the case for one of our pen-tested systems.

In addition to basic cyber security concerns, the advanced measurement capabilities of smart meters makes them a potential threat to privacy if used in an unrestricted manner. This is due to their ability to implement Non-Intrusive Load Monitoring (NILM), which can disaggregate the loads exerted by the individual appliances in a house from the net load recorded at the electric meter [13]. Hart posited NILM's use as a means of surveillance over activities that are normally considered within the sanctity of the home [12]. More recently, Lisovich et al. showed that the appliance information extracted by NILM is useful to recover some information about occupant behavior [21]. While this paper is limited to AMI related concerns, we mention that attacks on sensors in the grid's core distribution network have also been considered [22], along with the necessary conditions for such attacks to lead to large scale cascading failures [19].

4. ARCHETYPAL ATTACK TREES

Having reviewed the general architecture of smart metering systems, we may now construct archetypal trees that describe attacks in a broad sense that is applicable to any system within the architecture. An archetypal tree is an attack tree that is general enough to be applicable to all systems of a given architecture. As with a regular attack tree, the root of an archetypal tree is a single adversarial goal. This goal is repeatedly broken down into subgoals that describe the individual conditions that must exist to reach the root goal. Unlike a regular attack tree, the leaf nodes of the archetypal tree are not targeted at a specific system. Instead, the leaves constitute the points to which concrete trees are grafted. It is thus critical that they be selected to clearly define the boundary between broad architectural goals and vendor-specific goals. While this is somewhat of an art rather than a science, we have devised a set of criteria to aid us in differentiating between archetypal and concrete goals. If any of the following are true of a goal during the construction of an archetypal tree, then it becomes a leaf node, to which a concrete tree can be grafted.

1. *The goal targets a component whose implementation is vendor-specific.* An example of such a component is the meter LAN. While an archetypal tree can prescribe an attack on a meter LAN, the attack can not be specific to any particular LAN media.
2. *The goal may be hindered by the presence of a vendor-specific protection mechanism.* The addition of any subgoals for circumventing vendor-specific protection mechanisms is by definition not archetypal. Such details must be described in the concrete tree. An example of this can be seen in the following section on energy fraud (Section 4.1), where nothing general is known about the protection mechanisms present at the collector's link to the backhaul network.

If a subgoal does not meet these conditions, it is broken down. In the following sections, we provide justification for extending or terminating a given subgoal where instructive.

4.1 Energy Fraud

For our initial pen-testing efforts [24], we constructed an archetypal tree for energy fraud (shown in Figure 1). It is described here so that it may be instantiated later. We define energy fraud as any tampering with the metering infrastructure that leads to a customer not being billed for some energy consumed. (Note that in this particular archetypal tree, we do not consider using energy fraud to artificially inflate a victim's bill.) In AMI, fraud may be committed in the field by modifying the recorded energy usage before it is read by the utility. Known methods for fraud in electromechanical meters include interfering with the meter's sensors using magnets and rewinding usage gauges by inverting the meter in the socket (thereby reversing current flow through the meter).

Smart meters, present new opportunities for tampering with usage data. As shown in the first level of subgoals in the example tree, this can be done in three places (a) in the meter's low-level components, (b) the meter's long-term storage, and (c) in transmission to the utility. The archetypal attacks in this tree, as in the others, are labeled as $TX.Y$, where T is a letter specific to the tree, X is the index of the subtree below the root to which the attack belongs, and Y is the index of the attack within that subtree. Starting with the physical attacks in subtree a, there are two means to interrupt a smart meter's physical measurement of usage. A1.1 simply requires that the meter is removed from the path of current flow, and A1.2 that it be reversed in

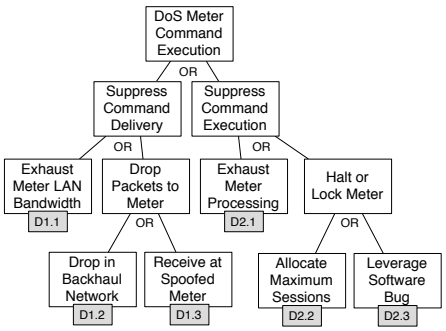


Figure 4: Archetypal tree for Denial of Service.

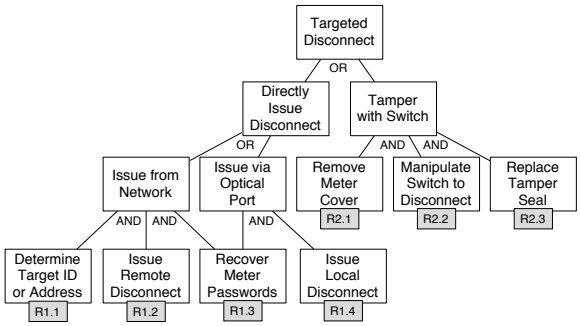


Figure 5: Archetypal tree for targeted disconnect.

its socket. As described in section 3.1, virtually all smart meters will log and report both of these events (power cycle and reverse energy flow respectively). Thus, in the archetypal level, we already recognize that the log messages will need to be cleared of these events. As a final note on physical attacks, because obtaining physical access to the meter is specific to a *particular installation*, we do not consider this prerequisite in either the archetypal or concrete trees. This does not matter for the case of fraud because it is assumed that the adversary already has access to her own meter.

Modifying logs and usage in meter storage is the goal of subtree *b*. This can be achieved in one of two ways. Either the meter's administrator password can be obtained and used to clear the log files: A2.1 AND A2.2, or the physical storage device may be tampered without interfering with the meter. As this is an archetypal tree, the implementation of the storage is left unmentioned.

The strategies for forging usage data on the wire are shown in subtree *c*. The interception of network communications is assumed to be necessary both for the purposes of understanding the meter's protocol stack, assuming it is nonstandard, and for interposing one's self in the communication path with the utility. In the archetypal tree, we ignore over which network (meter LAN or backhaul) the interception occurs, as well as any potential protection mechanisms. Along with A3.1, the adversary must either hijack a session between the meter and utility (A3.2) or impersonate a meter for the entire session (A3.3).

4.2 Denial of Service

This section considers DoS attacks that prevent meters from acting on commands such as usage queries, firmware upgrades, and remote disconnects. This is a realistic adversary goal. For example, if the retrieval of meter log files can be prevented for a sufficient period of time, a suspicious event such as a meter power cycle can be erased when the logs roll over with benign events.

The archetypal tree for meter DoS against meter command execution is shown in Figure 4. The adversary has two choices for a general strategy, either prevent the command from reaching the meter, or prevent its execution on the meter. The former can be achieved either through network resource exhaustion, or by tampering with the routing of packets away from the meter. As the LAN media is system specific, we do not break this subgoal down any further in the archetypal tree. A potentially more practical strategy is to drop traffic destined for the meter. This may either be done at a link or routing layer (D1.2) or at the transmission layer (D1.3). The latter seems like the more reasonable method, as dropping a packet at an intermediate hop will result in a retransmission by a higher layer.

The second strategy for command DoS prevents the meter from executing a command once it is received. An extremely simplistic method for doing this is to exhaust the meter's input processing capability (D2.1). This could be done either from the backhaul network or meter LAN. While effective, this type of attack is not covert, and cannot guarantee the command will fail. A more failsafe approach would be to put the meter into an unresponsive state. This may be done through interactions that exhaust a particular system resource, e.g. allocating and maintaining the maximum allowed number of open connections (D2.2), or by leveraging a firmware bug causing a system hang (D2.3).

4.3 Targeted Disconnect of Electrical Service

Most meter vendors include remote disconnect functionality in their meters. The ability to disconnect a target's power can cause at best, inconvenience and in worse scenarios, financial or physical harm depending on the setting. As described earlier, remote disconnect systems consist of a physical switch that breaks the current flowing to the house, and a set of remote commands to operate this switch. The archetypal tree for this attack is shown in Figure 5.

The ideal case for an adversary would be to issue the disconnect command remotely. Doing this requires at least that the ID be known for the target device (R1.1), and that its administrator password has been recovered (R1.3). Notice that this is the second archetypal tree with a leaf node requiring meter passwords to be recovered. This illustrates a secondary usefulness of attack trees: they act as a reference for quickly mapping security flaws to the adversarial goals they enable.

We reason that the disconnect functionality will be accessible through the optical ports on most systems because optical port functionality needs to contain at least the network functionality to allow the meter to function in the event that it is not network accessible, e.g. the meter's network card is malfunctioning. This is the basis of archetypal attacks R1.3 and R1.4.

Finally, physical access to a meter may also be useful for manipulating the disconnect switch, be it by mechanical or electrical means (R2.2). From experience, we have found that virtually all smart meters use the same tamper seal [1]. We have contacted the manufacturer of these seals and con-

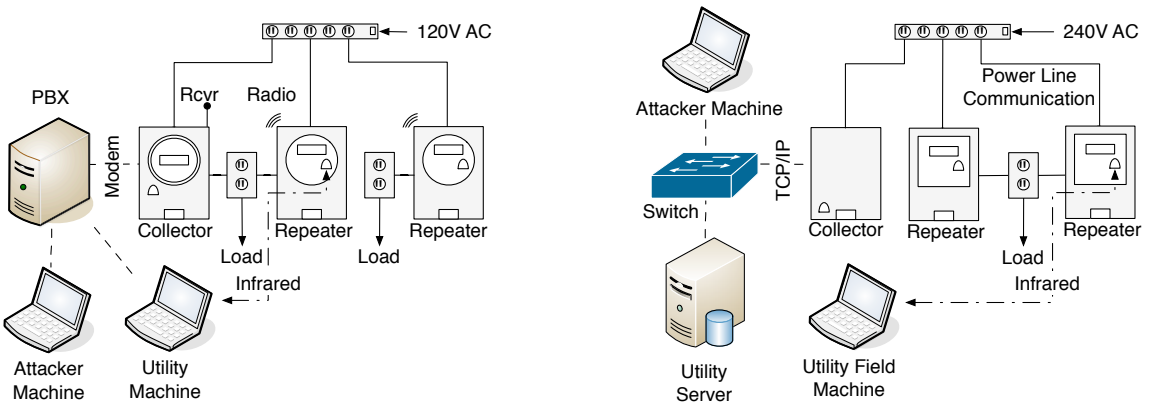


Figure 6: The two SUTs used in our experiments. In $S1$ (A), the collector also functions as a meter, and relays data from a wireless mesh LAN to a telephone network backhaul. $S2$ (B) uses a dedicated device as a collector to relay data between a PLC network and an Internet connection to the utility.

firmed that there are no limitations on the text which we could have embossed on the flag.

5. SYSTEMS UNDER TEST

This section details the two Systems Under Test (SUTs) that have been the subject of our penetration testing[1]. We will denote the two systems as $S1$ and $S2$. Besides the meters themselves, this section covers the additional components needed to run utility-end software and to network meters with the utilities. In describing the two systems, we will refer to the *utility machine* or *utility server* to mean a Microsoft Windows-based PC or laptop computer running software for meter management. We found that Windows by far the most common choice of utility-end operating system across vendors. The *attacker machine* is used to represent our machine used for various pen-testing purposes. In practice, this could be any machine within network reachability of a meter that is controlled by an adversary.

The general environment for both systems is identical. Both SUTs consist of several repeaters and a single collector, the main difference being that in $S2$, the collector does not function as a meter itself. We constructed sockets to allow the meters in our lab to function using wall socket power. The meters in $S1$ are able to run on 120V AC at 60 Hz, while the meters in $S2$ require a 240V step up transformer. A simple load was exerted by a small synchronous motor and measured to check the proper installation of each meter.

5.1 S1 Specifics

An overview of $S1$ is given in Figure 6.A. In $S1$, utilities communicate with meters via Public Switched Telephone Service. For obvious security reasons, we were unable to directly connect our collector to the telephone network. Instead, an Asterisk [29] based private branch exchange (PBX) on an x86 Linux machine provided call routing between the collector and utility machine. The PBX routes calls according to a table called the *dial plan*. The attacker machine sits on the PBX along with the meter and utility machine. Calls to the meter can be routed to the attacker machine by modifying the PBX dial plan. The ability to perform such rerouting using a commodity system was instrumental in our instantiation of the energy fraud attack for $S1$.

For all communication, the utility machine initiates communication with collector meters, with the exception of alarm conditions such as outage management or potential intrusions, in which case the meter preemptively contacts the utility. We augmented the utility machine with a modem monitor for analyzing the telephone protocol. What we quickly found is that it largely conforms to the ANSI C12.21 standard for telephone modem communication with meters. After this, the monitor was only needed to understand the occasional deviations from the standard.

The PSTN backhaul link at the collector is guarded by an "intrusion detection" mechanism. The purpose of this mechanism is to prevent both active and passive attacks from telephony devices connected on the same link as the collector, i.e. via a line splitter. The intrusion detection mechanism will immediately terminate a call from the utility if another device on the line goes off the hook. When a device goes off the hook, it receives a dial tone and voltage via an onboard component called the Foreign Exchange Office (FXO). All endpoint devices in a telephone network use an FXO. The dial tone and voltage are supplied from the other end of the line by the Foreign Exchange Service (FXS), usually implemented by the phone company. Because the meter can detect when another device is receiving a voltage and dial tone, it can terminate its current call.

The main operation of concern in $S1$ is the diagnostic protocol between the meter and utility. This protocol can perform many functions from a simple meter reading, to a full check of every parameter set in the meter. For the purposes of energy fraud we are mainly concerned with how this protocol performs energy usage readings. The utility initiates a diagnostic by calling a collector, resulting in the collector responding with an identification message. An authentication round is then carried out according to the default scheme specified by ANSI C12.21 (ANSI X3.92-198) [3]. If authentication is successful, the utility will probe the meter for

[1]We do not reveal vendor identities here, as we are already in contact with them, and both SUTs are already deployed in the US and Europe.

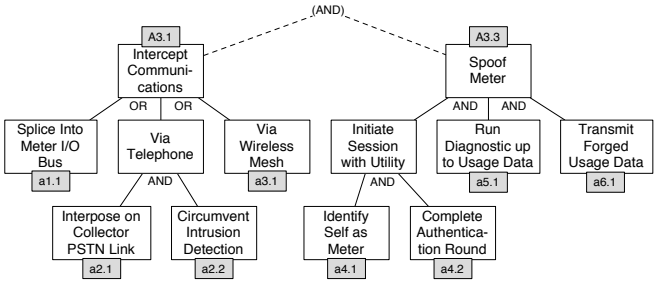


Figure 7: The concrete trees for energy fraud in $S1$.

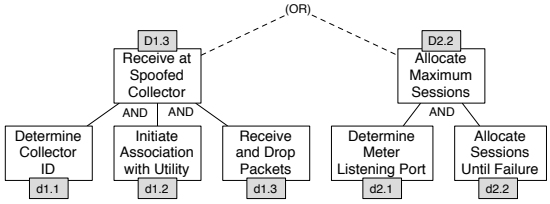


Figure 8: The concrete trees for DOS in $S2$.

some variable number of parameters, after which the current net usages are read. This is the point in the protocol where a usage forgery must occur. The remainder of the protocol consists of potentially more parameter queries, and finally a goodbye message. The meter LAN, a wireless mesh operating in the 900 MHz band, is currently under evaluation.

5.2 S2 Specifics

Our testbed for $S2$ is shown in Figure 6.B. The main differences from $S1$ are the backhaul and meter LAN protocols, and the collector, which does not function as a meter in $S2$. Upon initial inspection, one notices that $S2$ is more accessible to remote attacks due to the use of an Internet-based backhaul. This fact becomes useful when instantiating a concrete tree for DoS against meter command execution. The meter LAN uses a proprietary protocol that requires special equipment to analyze.

Though the application layer protocol between the utility and collector is proprietary, two thing are clear from initial inspection. First, an initial association between the two is started by the collector, and each subsequent command execution is started by the utility. This suggests that both directions should be considered when designing a concrete DoS attack. Second, in the initial association, the collector transmits its unique ID number and associated network address in the clear to the utility. Thus, knowing this ID for a target collector may be useful in a DoS attack.

6. CONCRETE ATTACK TREES

Concrete attack trees function as a guide for penetration testing a specific system. As with the archetypal trees, we use basic guidelines to determine when a concrete tree is specific enough. Any details not elaborated in the concrete tree must either already be known about the system, or must be discovered during pen-testing. In constructing the concrete trees for fraud, DoS, and targeted disconnect, we use the following two rules:

1. *A goal should be a leaf if it is achievable completely by known means in the system.* This is the simplest case as no additional pen-testing is required. Several leaves in the concrete DoS tree are of this type.
2. *A goal should be a leaf if no vulnerability is yet known that would allow it to be executed.* At this point, determining the existence of a vulnerability enabling the goal becomes the job of penetration testing.

We instantiate concrete trees for the three adversarial goals for $S1$ and $S2$ below. The root of each concrete tree shares a reference number with a leaf in one or more archetypal attack trees to which it may be grafted. We instantiate fraud and targeted disconnect for $S1$, and DoS for $S2$[2].

6.1 Energy Fraud in S1

The archetypal attack tree for energy fraud presented three broad strategies: tampering with the measurement process, tampering with the recorded usage in meter storage, and tampering with the usage data in transmission. For our first attempt to implement a fraud attack in $S1$, we chose the third strategy because of its relatively low invasiveness and our understanding of the backhaul network operation. This strategy terminated in three archetypal attacks: a mandatory requirement of being interposed on the backhaul link (A3.1), and the option of either performing a man in the middle attack (A3.2) or meter spoofing (A3.3). After evaluating the ANSI C12.21 specification via trace of $S1$'s telephony-based diagnostic protocol, we determined that meter spoofing was more straightforward. Thus, to complete the goal of fraud in $S1$, we must instantiate and execute concrete trees for archetypal attacks A1.1 and A1.3. Both concrete trees are shown in Figure 7.

Archetypal attack A3.1 requires that the adversary be interposed somewhere on the path between the meter's networking interface card (NIC) and the utility. In one extreme end, this may be achieved by directly tampering with the communications bus on which the NIC resides (a1.1). Two more likely places are the mesh network (a3.1), and the telephone backhaul (a2.1). For the latter, the additional prerequisite of bypassing the "intrusion detection" mechanism is necessary (a2.2).

The second archetypal attack for energy fraud requires meter spoofing. This calls for three steps to successfully deliver forged usage data as part of $S1$'s diagnostic protocol. First, the spoofing device must initiate a new diagnostic session with the utility. This will require first identifying itself as the expected meter (a4.1), and second, completing the authentication round (a4.2). Once the session is established, the spoofing device must answer all diagnostic queries up to the forged demand (a5.1), and finally, insert the forged demand value (a6.1). The remainder of work to realize these attacks is achieved by pen-testing as described in section 7.

6.2 Denial of Service in S2

Unlike the two concrete trees for energy fraud, the root nodes of the two for DoS are combined by disjunction in the archetypal tree. Thus, fulfilling the requirements of either

[2]While there are a large number of attempted attacks, we find the ones described here to be the most instructive.

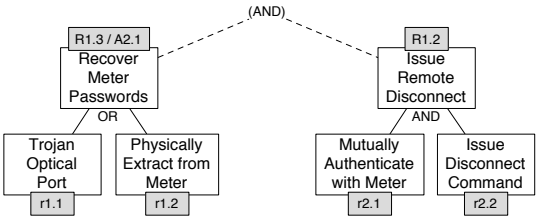


Figure 9: The concrete trees for targeted disconnect $S1$.

tree is sufficient for achieving denial of command execution. Recall that there are two options because communication in $S2$ may be initiated by both the collector and the utility at different points in time. The first tree (D1.3) requires another device to spoof the collector node in order to receive any commands destined for meters and drop them en route. This requires first the necessary reconnaissance to determine the collectors network ID (d1.1), and to establish a new session with the utility using that ID (d1.2). Finally, the spoofed collector can receive and drop commands from the utility (d1.3). All three of these are leaves in the concrete tree because they are achievable using known actions within the system.

The other option for DoS against utility command execution is to allocate a maximum number of sessions in the meter (D2.2). First, it must be determined on which port the meter listens for commands (d2.1). If this is possible, an attempt may be made to open multiple sessions on this port in an attempt to exhaust either memory or OS resources in the meter (d2.2). Both concrete attacks are leaf nodes because pen-testing of $S2$ is needed to determine how they may be executed in practice.

6.3 Targeted Disconnect in S1

The final concrete attack tree analyzed here is for the disruption of electrical service. As an adversary would ideally want to execute this attack remotely, we chose archetypal attacks R1.1 - R1.3 for instantiation. In $S1$, the meter ID is printed on the front of each meter, making R1.1 achievable by visual inspection. The concrete trees for R1.2 and R1.3 are shown in Figure 9.

Two strategies are feasible for meter password recovery in $S1$ (R1.3). If the optical port can be physically monitored, then the password can be obtained upon the next visit by the utility (r1.1). Alternatively, if the contents of meter storage can be extracted, the password may be recoverable, though potentially only in a hashed format (r1.3). As both of these are physical attacks, they may only be used to recover a password from a single meter. This would normally be a limiting factor in the impact of an attack against $S1$, but we observe that its architecture encourages utilities to use the same password for a large number of meters. In the administrative utility-end software, a single password set (consisting of a read-only and administrative user) is chosen for a template program that is pushed to the meters at configuration time. This makes it very tedious to create a different program template for each meter. A brute force guessing attack is not considered, as the maximum length of a password in $S1$ is well over ten bytes. The final archetypal attack needed is the issuance of the command to the target meter. This requires that the known password be used in the mutual authentication round (the same as that used in $S1$'s diagnostic protocol) (r2.1). Once authenticated, the command can be issued (r2.2).

7. RESULTS

We now turn to the results of the penetration testing to achieve each goal as summarized in Table 1.

7.1 Energy Fraud by Forged Usage Data

The energy fraud attack in $S1$ works as follows. First, an adversarial device is interposed on the PSTN link from a collector (a2.1) so as not to trigger the intrusion detection mechanism (a2.2). This was achieved by interposing our PBX on the line. Recall that the purpose of the intrusion detection feature is to protect meter communication in situations where the link to the PSTN is shared with that already present in a house. The PBX is used to route incoming calls to the meter to a laptop computer that impersonates the meter using a Python program we wrote. This is sufficient for circumventing the intrusion detection mechanism for two reasons. First, routing a call to the laptop need not involve the meter at all. Second, if the PBX is used for the purposes of eavesdropping on communication between the meter and utility, it cannot be detected by the intrusion detection mechanism that can only sense other FXOs on the line (as described in section 5.1). Thus two requirements of A3.1 are satisfied.

Once the adversarial laptop has been contacted by the utility, it must identify itself as the target meter (a4.1) and complete the authentication round (a4.2). This was possible without knowing the meter's password, which is used to derive the key for the authentication protocol. Spoofing meter identification only required using the ID which was printed on the meter's nameplate. Completing the authentication round without knowing the password required one observation about the protocol: the meter generates the nonce used for mutual authentication, but nonces are not tracked by the utility's server. Thus, a replayed nonce is sufficient for replaying the remainder of the authentication protocol. What was not initially obvious was that the meter places the nonce in a special field as part of the identification round. Thus, replaying both the identification and authentication rounds of ANSI C12.21 is sufficient for spoofing the meter during a diagnostic.

The remaining protocol up to forged demand insertion may also be replayed in this manner, satisfying (a5.1). The final task towards energy fraud is inserting a forged net usage value into the diagnostic. This requires adding two additional pieces of information along with the numerical usage value. First, a one byte checksum of the value is placed in the application-layer header, and second a CRC is placed in the MAC layer header, again as specified in ANSI C12.21.

Table 1: Summary of concrete attacks and discovered vulnerabilities for each adversarial goal.

Ref.	Description	Enabling Feature or Vulnerability
	Energy Fraud in $S1$	
a2.1	Interpose between utility and collector	Telephone line may be accessible.
a2.2	Defeat modem intrusion detection	The mechanism cannot detect an FXS.
a4.1	Identify self as meter	A meter's ID is printed on its faceplate.
a4.2	Complete authentication round	Lack of nonce-tracking allows replayed authentication.
a5.1	Run diagnostic up to usage data	Protocol is standardized.
a6.1	Transmit forged usage data	Usage data is not integrity protected.
	Denial of Service in $S2$	
d1.1	Determine collector ID	The ID is transmitted in the clear.
d1.2	Initiate association with utility	Initialization uses a simple `init` message.
d1.3	Receive and drop packets	The utility uses the IP address of the initiator of the most recent association.
d2.1	Determine meter listening port	The collector is responsive to port scanning.
d2.2	Allocate sessions until failure	The collector does not handle many sessions robustly.
	Targeted Disconnect in $S1$	
r1.2	Physically extract passwords	Passwords are stored in the clear in EEPROM storage.
r2.1	Mutually authenticate with meter	The encryption key is derived from passwords.
r2.2	Issue disconnect command	Administrative software is commercially available.

7.2 Denial of Service Against Command Execution

Two concrete attack trees were previously introduced for Denial of Service against the execution of utility commands by meters in $S2$. The first assumed that an association could be formed between the utility and a device impersonating a collector (d1.1,d1.2). At this point, the fake collector could simply drop all commands issued by the utility (d1.3). The initial association with the utility is initiated by an `init` sent by the collector. This message, which is transmitted in the clear, contains the unique serial number used to identify the collector. The utility assumes that the source IP address of the `init` message is the collector. Any device may submit an `init` message to the utility, but will not be able to establish a secure channel without knowing the collector's symmetric key. This does not prevent the DoS attack however, as receiving the `init` message causes the utility to drop its previous association with the real collector. After this, the collector will only attempt to create a new association if it is rebooted or if some alarm condition occurs such as a power outage or potential physical tampering. A subsequent second forged `init` would suffice to immediately break this association.

The other concrete attack tree in this category is based on the idea that the collector has a maximum number of sessions which can be reached (D2.2). In practice, finding the port on which a collector listens for utility requests (d2.1) is done using the `nmap` [2] utility to perform a port scan of the collector. What we found was that while attempting to open many concurrent TCP connections on the collector's listening port, the collector would become unresponsive after fewer than ten such connections. If continual attempts at establishing new connections were made at the rate of once per ten seconds, the collector remains unresponsive, and the utility-end server is unable to complete any commands on that collector, thus satisfying d2.2.

Under the category of DoS, we do have one result that was found completely independently of the methodology presented in this paper. The use of a software fuzz tester [16] found that the collector was vulnerable to crashing while processing malformed packets. While we had not planned on systematically exploring methods for leveraging software bugs (D2.3), the use of fuzz testing would make a viable addition to the archetypal tree.

7.3 Targeted Disconnect

The final result we explore is the application of concrete trees R1.2 and R1.3 to disrupting electric service at a target meter by subverting its remote disconnect feature. We were unable to verify the efficacy of this attack due to fact that our $S1$ meters do not include the optional physical disconnect switch. However, we reason by inspection that the attack is possible. The first step needed to issue the remote disconnect command is password recovery (R1.3). After some experimentation, we found that concrete attack (r1.2) is possible. By desoldering a small SPI-based EEPROM memory chip from the $S1$ collector's radio card, we were able to extract the plaintext password. While this is a potentially dangerous operation, given that the same password may be used throughout a deployment, the payoff is high. Upon discovering that the meter passwords may be extracted from memory, a check of the archetypal trees reveals that A2.1 from Figure 1 is also satisfied, enabling several alternate strategies for energy fraud. This demonstrates the usefulness of archetypal attack trees in mapping newly discovered vulnerabilities to adversarial goals.

Once the password is recovered, it must be used to perform the default C12.21 authentication function with the target meter (r2.1). This authentication is a keyed hash based on the DES cipher, and thus requires a DES key. An internet search revealed that one distributor of $S1$ had placed the manual for the utility-end software in a publicly accessible directory. The manual revealed the fact that the first eight bytes of the password are used to derive a DES key. This is done using an unknown obfuscation method. The easiest procedure to use the recovered passwords for authenticating to the target meter would be to obtain the utility-end software, which can be purchased from third party distributors, and provide it the password to issue the discon-

nect command (r2.2). Otherwise, a degree of reconnaissance and reverse engineering will be necessary to determine the obfuscation method.

8. CONCLUSIONS

In this paper, we have investigated a technique for evaluating the security of the myriad of devices being deployed into the AMI. We have shown that we can leverage focused penetration efforts in one vendor to others, and explored where such evaluations must focus solely on the unique artifacts of a system under test. In so doing, this work has sought not only to streamline security analysis, but also to ensure greater and more consistent coverage of potential attacker goals and methods.

Yet there is much work left to be done. Government agencies such as the NIST and the Federal Energy Regulation Commission (FERC) continue to provide the essential guidelines for the design and maintenance of AMI *security infrastructure*. Complementary efforts at codifying penetration testing of AMI such as the one documented in this paper are essential to the future reliability of electric power grids. In the future, we will expand the base of attacker goals and associated trees, as well as extend this work to other vendor devices. It is through these collected efforts that we hope to garner a broad view of the security issues in AMI, and ultimate positively influence the safety of smart grid.

9. REFERENCES

[1] B.T. Aluminum Tamper Seal. `http://www.brooksutility.com/catalog/product-detail.asp?ID=302`.

[2] Nmap Reference Guide. `http://nmap.org/book/man.html`.

[3] American National Standards Institute. ANSIX3.92-198 Data Encryption Algorithm, 1981.

[4] American National Standards Institute. C12.18 Protocol Specification for ANSI Type 2 Optical Port, 2006.

[5] A. Brothman, R. D. Reiser, N. L. Kahn, F. S. Ritenhouse, and R. A. Wells. Automatic Remote Reading of Residential Meters. *IEEE Transactions on Communication Technology*, 13(2):219 – 232, 1965.

[6] E. Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley, 2005.

[7] W. Enck, P. Traynor, P. Mcdaniel, and T. L. Porta. Exploiting Open Functionality in SMS-capable Cellular Networks. In *Proceedings of the 12th ACM Conference on Computer and Communication Security (CCS)*, pages 393–404. ACM Press, 2005.

[8] C. A. Ericson, II. Fault Tree Analysis — A History. In *Proceedings of the 17th International System Saftey Conference*, 1999.

[9] K. Fehrenbacher. Smart Meter Worm Could Spread Like A Virus. `http://earth2tech.com/2009/07/31/smart-meter-worm-could-spread-like-a-virus/`.

[10] M. Gegick and L. Williams. Matching attack patterns to security vulnerabilities in software-intensive system designs. In *SESS '05: Proceedings of the 2005 workshop on Software engineering for secure systems—building trustworthy applications*, pages 1–7, New York, NY, USA, 2005. ACM.

[11] M. Goldberg. Measure Twice, Cut Once. *IEEE Power and Energy Magazine*, May/June 2010.

[12] G. W. Hart. Residential Energy Monitoring and Computerized Surveillance via Utility Power Flows. *IEEE Technology and Society Magazine*, June 1989.

[13] G. W. Hart. Nonintrusive Appliance Load Monitoring. *Proceedings of the IEEE*, 2004.

[14] G. Hoglund and G. McGraw. *Exploiting Software: How to Break Code*. Addison Wesley, 2004.

[15] D. Hull. PG&E details technical problems with SmartMeters. `http://www.siliconvalley.com/news/ci_14963541`, April 2010.

[16] Infigo.hr. Infigo FTPStress Fuzzer. `http://www.infigo.hr/en/in_focus/tools`.

[17] R. Kelley and R. D. Pate. Mesh Networks and Outage Management. White Paper, September 2008.

[18] C. S. King. The Economics of Real-Time and Time-of-Use Pricing For Residential Consumers. Technical report, American Energy Institute, 2001.

[19] R. Kinney, P. Crucitti, R. Albert, and V. Latora. Modeling cascading failures in the North American power grid. *The European Physical Journal B - Condensed Matter and Complex Systems*, 46(1):101–107, July 2005.

[20] N. Lewson. Smart meter crypto flaw worse than thought. `http://rdist.root.org/2010/01/11/smart-meter-crypto-flaw-worse-than-thought`.

[21] M. A. Lisovich, D. K. Mulligan, and S. B. Wicker. Inferring Personal Information from Demand-Response Systems. *IEEE Security and Privacy*, 8:11–20, 2010.

[22] Y. Liu, P. Ning, and M. K. Reiter. False Data Injection Attacks against State Estimation in Electric Power Grids. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, November 2009.

[23] P. McDaniel and S. McLaughlin. Security and Privacy Challenges in the Smart Grid. *IEEE Security & Privacy Magazine*, May/June 2009.

[24] S. McLaughlin, D. Podkuiko, and P. McDaniel. Energy Theft in the Advanced Metering Infrastructure. In *Proceedings of the 4th International Workshop on Critical Information Infrastructure Security*, 2009.

[25] R. Meritt. Stimulus: DoE readies $4.3 billion for smart grid. *EE Times*, February 2009.

[26] A. H. Rosenfeld, D. A. Bulleit, and R. A. Peddie. Smart Meters and Spot Pricing: Experiments and Potential. *IEEE Technology and Society Magazine*, March 1986.

[27] B. Schneier. Attack Trees. *Dr. Dobb's Journal*, December 1999.

[28] A. Takanen, J. DeMott, and C. Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House Publishers, 2008.

[29] The Asterisk Project. Asterisk open source pbx. `http://www.asterisk.org`.

[30] The Smart Grid Interoperability Panel – Cyber Security Working Group. Smart grid cyber security strategy and requirements draft nistir 7628, February 2010.

[31] P. Traynor, M. Lin, M. Ongtang, V. Rao, T. Jaeger, P. McDaniel, and T. La Porta. On Cellular Botnets: Measuring the Impact of Malicious Devices on a Cellular Network Core. In *Proceedings of the 16th ACM Cnference on Computer and Communications Security (CCS)*, pages 223–234, New York, NY, USA, November 2009. ACM.

[32] W. Vesely, F. Goldberg, N. Roberts, and D. Haasl. *Fault Tree Handbook*. U.S. Nuclear Regulator Commission, 1981.

[33] K. Zetter. Security Pros Question Deployment of Smart Meters. *Threat Level: Privacy, Crime and Security Online*, March 2010.

Network Intrusion Detection: Dead or Alive?

[Classic Paper]

Giovanni Vigna
Department of Computer Science
University of California, Santa Barbara
vigna@cs.ucsb.edu

ABSTRACT

Research on network intrusion detection has produced a number of interesting results. In this paper, I look back to the NetSTAT system, which was presented at ACSAC in 1998. In addition to describing the original system, I discuss some historical context, with reference to well-known evaluation efforts and to the evolution of network intrusion detection into a broader field that includes malware detection and the analysis of malicious behavior.

Keywords

Intrusion Detection, Network Security

1. INTRODUCTION

Network intrusion detection systems (NIDSs) have evolved from their academic beginnings into mainstream commercial products, and network intrusion detection is now considered a "mature technology." From the early network-based systems (such as EMERALD [13], NSM [3], Bro [11], and NetSTAT [16]), dozens of network-based systems have been proposed in research and many have transitioned to the commercial world to become products (see, for example, Snort [14], which is the most popular open-source network intrusion detection system today).

Even though network intrusion detection is considered a mature technology and research in this field is sometimes considered "dead," network attacks are still prevalent, large-scale abuse of network resources are an everyday reality, and sophisticated attacks seem to be able to easily bypass commercial intrusion detection systems. So what happened to network intrusion detection?

In this paper, I look back to some early research in network intrusion detection, namely the NetSTAT system, which was presented at ACSAC in 1998 [16]. I describe the system in Section 2 and present some interesting contributions which are still unmatched by the current state-of-the-art tools.

In Section 3, I discuss how, in the late nineties, there was a push to compare and evaluate the intrusion detection research being performed at the time, which culminated in the MIT Lincoln Laboratory's intrusion detection system evaluation effort. Even though the results of this effort were criticized and misused, they still represent one of the most systematic and interesting attempts to measure, compare, and even stimulate research in security.

Then, in Section 4, I describe some of the shortcomings that gave network intrusion detection a bad name, but I also discuss how the lessons learned in developing intrusion detection systems have been taken into account in shaping a larger research field, involved with the detection of compromises at many levels.

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

2. THE NETSTAT SYSTEM

The NetSTAT system was a network-based intrusion detection system. NetSTAT extended the state transition analysis technique (STAT) [4] to network-based intrusion detection in order to represent attack scenarios in a networked environment. However, unlike other network-based intrusion detection systems that monitored a single sub-network for patterns representing malicious activity, NetSTAT was oriented towards the detection of attacks in complex networks composed of several sub-networks. In this setting, the messages that are produced during an intrusion attempt may be recognized as malicious only in particular subparts of the network, depending on the network topology and service configuration. As a consequence, intrusions cannot be detected by a single component, and a distributed approach is needed.

The NetSTAT approach models network attacks as state transition diagrams, where states and transitions are characterized in a networked environment. The network environment itself is described by using a formal model based on hypergraphs [1, 15].

The analysis of the attack scenarios and the network formal descriptions determines which events have to be monitored to detect an intrusion and where the monitors need to be placed. In addition, by characterizing in a formal way both the *configuration* and the *state* of a network it is possible to provide the components responsible for intrusion detection with all the information they need to perform their task autonomously with minimal interaction and traffic overhead. This can be achieved because network-based state transition diagrams contain references to the network topology and service configuration. Thus, it is possible to extract from a central database only the information that is needed for the detection of the particular modeled intrusions. Moreover, attack scenarios use assertions to characterize the state

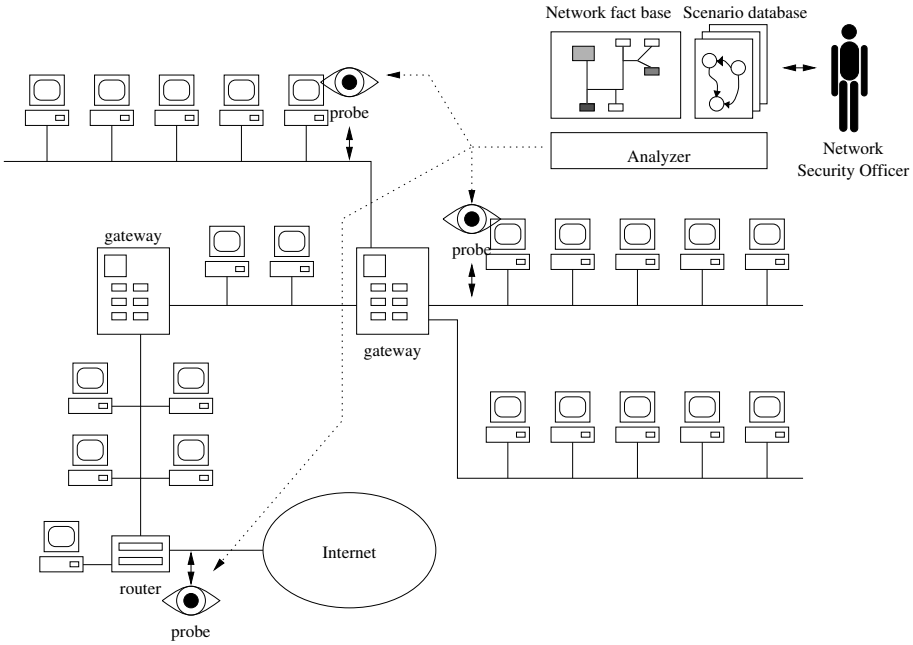


Figure 1: The NetSTAT architecture.

of the network. Thus, it is possible to automatically determine the data to be collected to support intrusion analysis and to instruct the detection components to look only for the events that are involved in run-time attack detection. This solution allows for a lightweight, scalable implementation of the probes and focused filtering of the network event stream, delivering more reliable, efficient, and autonomous components.

2.1 Architecture

NetSTAT is a distributed application composed of the following components: the network fact base, the state transition scenario database, a collection of general-purpose probes, and the analyzer. A high-level view of the NetSTAT architecture is given in Figure 1.

2.1.1 Network Fact Base

The network fact base component stores and manages the security-relevant information about a network. The fact base is a stand-alone application that is used by the Network Security Officer to construct, insert, and browse the data about the network being protected. It contains information about the network topology and the network services provided.

The network topology is a description of the constituent components of the network and how they are connected. The network model underlying the NetSTAT tool uses *interfaces*, *hosts*, and *links* as primitive elements. A network is represented as a hypergraph on the set of interfaces [15]. In this model, interfaces are nodes while hosts and links are edges; that is, hosts and links are modeled as sets of interfaces. This is an original approach that has a number of advantages. Because the model is formal, it provides a well-defined semantics and supports reasoning and automation. Another advantage is that this formalization allows one to model network links based on a shared medium (e.g., Ethernet) in a natural way, by representing the shared medium as a set containing all the interfaces that can access the communication bus. In this way, it is possible to precisely model the concept of network traffic eavesdropping, which is the basis for a number of network-related attacks. In addition, topological properties can be described in a simple way since hosts and links are treated uniformly as edges of the hypergraph.

The network model is not limited to the description of the connection of elements. Each element of the model has some associated information. For example, hosts have several attributes that characterize the type of hardware and operating system software installed. The reader should note that in this model "host" is a rather general concept. More specifically, a host is a device that has one or more network interfaces that can be the (explicit) source and/or destination of network traffic. For example, by this definition, gateways and printers are considered to be hosts. Links are characterized by their type (e.g., Ethernet). Interfaces are characterized by their type and by their corresponding link- and network-level addresses. This information is represented in the model by means of functions that associate the network elements with the related information.

The network services portion of the network fact base contains a description of the services provided by the hosts of a network. Examples of these services are the Network File System (NFS), the Network Information System (NIS), TELNET, FTP, "r" services, etc. The fact base contains a characterization of each service in terms of the network/transport protocol(s) used, the access model (e.g., request/reply), the type of authentication (e.g., address-based, password-based, token-based, or certificate-based), and the level of traffic protection (e.g., encrypted or not). In addition, the network fact base contains information about how services are deployed, that is, how services are instantiated and accessed over the network.

Figure 2 shows an example network. In the hypergraph

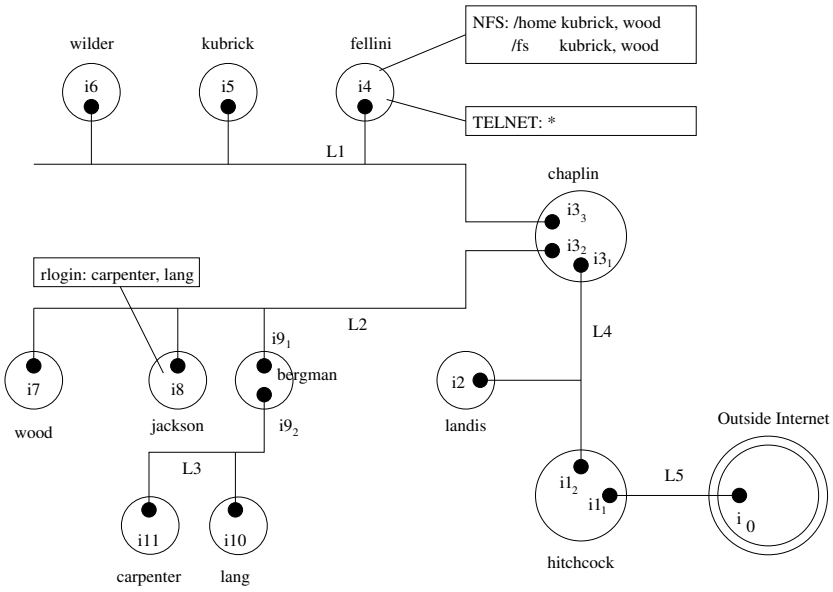


Figure 2: An example network.

describing the network, interfaces are represented as black dots, hosts are represented as circles around the corresponding interfaces, and links are represented as lines connecting the interfaces. The sample network is composed of five links, namely L_1, L_2, L_3, L_4, and L_5, and twelve hosts. Hereinafter, it is assumed that each interface has a single associated IP address, for example interface i_7 is associated with IP address a_7. The outside network is modeled as a *composite host* (the double circle in the figure) containing all the interfaces and corresponding addresses not in use elsewhere in the modeled network. As far as services are concerned, host `fellini` is an NFS server exporting file systems `/home` and `/fs` to `kubrick` and `wood`. In addition, `fellini` is a TELNET server for everybody. Host `jackson` exports an *rlogin* service to hosts `carpenter` and `lang`.

2.1.2 State Transition Scenario Database

The state transition scenario database is the component that manages the set of state transition representations of the intrusion scenarios to be detected.

The state transition analysis technique was originally developed to model host-based intrusions [4]. It describes computer penetrations as sequences of actions that an attacker performs to compromise the security of a computer system. Attacks are (graphically) described by using *state transition diagrams*. *States* represent snapshots of a system's volatile, semi-permanent, and permanent memory locations. A description of an attack has a "safe" starting state, zero or more intermediate states, and (at least) one "compromised" ending state. States are characterized by means of *assertions*, which are functions with zero or more arguments returning Boolean values. Typically, these assertions describe some aspects of the security state of the system, such as file ownership, user identification, or user authorization. *Transitions* between states are indicated by *signature actions* that represent the actions that, if omitted from the execution of an attack scenario, would prevent the attack from completing successfully. Typical examples of host-based signature actions include reading, writing, and executing files. For a complete description of the state transition analysis technique see [12]. For NetSTAT the original STAT technique has been applied to computer networks, and the concepts of state, assertions, and signature actions have been characterized in a networked environment.

States and Assertions.

In network-based state transition analysis the state includes the currently active connections (for connection oriented services), the state of interactions (for connectionless services), and the values of the network tables (e.g., routing tables, DNS mappings, ARP caches, etc). For instance, both an open connection and a mounted file system are part of the state of the network. A pending DNS request that has not yet been answered is also part of the state, such as the mapping between IP address `128.111.12.13` and the name `hitchcock`. For the application of state transition analysis to networks the original state transition analysis concept of assertion has been extended to include both *static assertions* and *dynamic assertions*.

Static assertions are assertions on a network that can be verified by examining the network fact base; that is, by examining its topology and the current service configuration. For example, the following assertion:

```
service s in server.services|
  s.name == "www" and
  s.application.name == "CERN httpd";
```

identifies a service `s` in the set of services provided by host `server` such that the name of the service is `www` and the application providing the service is the CERN http daemon[1]. As another example, the following assertion:

```
Interface i in gateway.interfaces|
  i.link.type == "Ethernet";
```

[1]The only (possibly) nonstandard notation used in the assertions is the use of "`|`" for "such that".

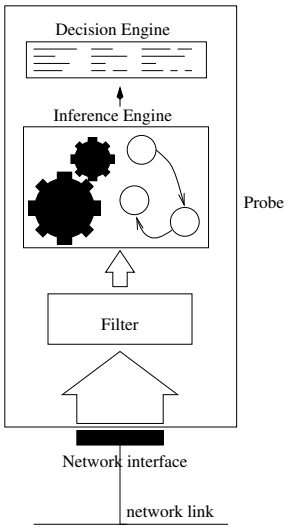


Figure 3: Probe architecture.

denotes an interface of a host, say `gateway`, that is connected to an Ethernet link.

These assertions are used to customize state transition representations for particular scenarios (e.g., a particular server and its clients). In practice, they are used to determine the amount of knowledge about the network fact base that each probe must be provided with during configuration procedures.

Dynamic assertions can be verified only by examining the current state of the network. One examples is `NFS-Mounted(filesys, server, client)`, which returns true if the specified file system exported by `server` is currently mounted by `client`. Another example is `ConnectionEstablished(addr1, port1, addr2, port2)`, which returns true if there is an established virtual circuit between the specified addresses and ports. These assertions are used to determine what relevant network state events should be monitored by a network probe.

Transitions and Signature Actions.

In NetSTAT, signature actions are expressed by leveraging an *event model*. In this model, events are sequences of messages exchanged over a network.

The basic event is the *link-level message*, or *message* for short. A link-level message is a string of bits that appears on a network link at a specified time. The message is exchanged between two directly-connected interfaces. For example the signature action:

```
Message m {i_x,i_y}|
  m.length > 512;
```

represents a link-level message exchanged between interfaces `i_x` and `i_y` whose size is greater than 512 bytes.

Basic events can be abstracted or composed to represent higher-level actions. For example, IP datagrams that are transported from one interface to another in an IP network are modeled as sequences of link-level messages that represent the intermediate steps in the delivery process. Note that the only directly observable events are link-level messages appearing on specific links. Therefore, the IP datagram "event" is observable by looking at the payload of one of the link-level messages used to deliver the datagram. For example, the signature action:

```
[IPDatagram d]{i_x,i_y}|
  d.options.sourceRoute == true;
```

represents an IP datagram that is delivered from interface `i_x` to interface `i_y` and that has the source route option enabled. This event can be observed by looking at the link-level messages used in datagram delivery along the path(s) from `i_x` to `i_y`. It is also possible to write signature actions that refer to specific link-level messages in the context of datagram delivery. For example, the signature action:

```
Message m in [IPDatagram d]{i_x,i_y}|
  m.dst != i_y;
```

represents a link-level message used during the delivery of an IP datagram such that the link-level destination address is not the final destination interface (i.e., the message is not the last one in the delivery process).

Events representing single UDP datagrams or TCP segments are represented by specifying encapsulation in an IP datagram. For example, the signature action:

```
[IPDatagram d [TCPSegment t]]{i_x,i_y}|
  d.dst == a_y and
  t.dst == 23;
```

denotes the sequence of messages used to deliver a TCP segment encapsulated into an IP datagram such that the destination IP address is `a_y` and the destination port is 23.

TCP virtual circuits are higher-level, composite events. A virtual circuit is identified by the tuple *(source IP address, destination IP address, source TCP port, destination TCP port)* and is composed of two sequences of TCP segments exchanged between two interfaces. Each of these two sequences defines a byte stream. The byte stream is obtained by assembling the payloads of the segments in the corresponding sequence, following the rules of the TCP protocol (e.g., sequencing, retransmission, etc.). The streams are denoted by `streamToClient` and `streamToServer`.

For example, the signature action:

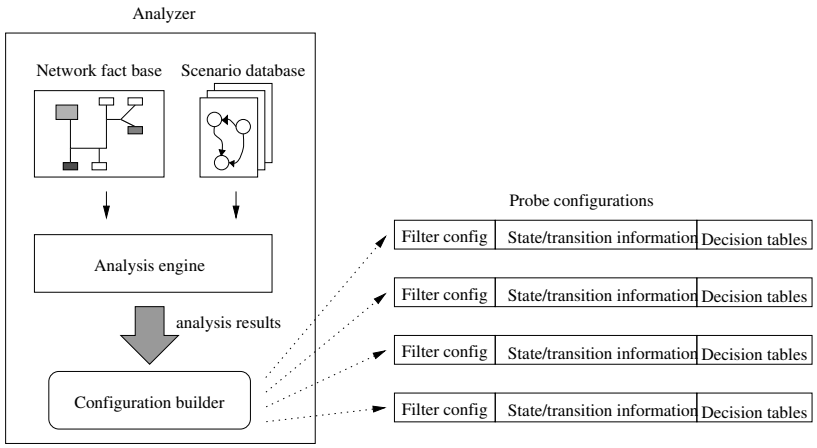


Figure 4: Analyzer architecture.

```
TCPSegment t in
          [VirtualCircuit c]{i_x,i_y}|
  c.dstIP == a_y and
  c.dstPort == 80 and
  t.syn == true;
```

denotes a segment that has the SYN bit set and belongs to a virtual circuit established between interfaces `i_x` and `i_y` and that has destination IP address `a_y` and destination port 80.

Events at the application level can be either encapsulated in UDP datagrams or can be sent through TCP virtual circuits. In the former case, the application-level event can be referenced by indicating the corresponding datagram and specifying the encapsulation. For example, the signature action:

```
[IPDatagram d
     [UDPDatagram u
          [RPC r]]]{i_x,i_y}|
   d.dst == a_y and
   u.dst == 2049 and
   r.type == CALL and
   r.proc == MKDIR;
```

represents an RPC request encapsulated in a UDP datagram representing an NFS command.

In the TCP virtual circuit case, application-level events are extracted by parsing the stream of bytes exchanged over the virtual circuit. The type of application event determines the protocol used to interpret the stream. For example, the following signature action:

```
[c.streamToServer [HTTPRequest r]]|
  r.method == "GET";
```

is an HTTP GET request that is transmitted over a TCP virtual circuit (defined somewhere else as `c`), through the stream directed to the server side[2].

2.1.3 Probes

The probes are the active intrusion detection components. They monitor the network traffic in specific parts of the network, following the configuration they receive at startup from the analyzer, which is described in the following section. Probes are general-purpose intrusion detection systems that can be configured remotely and dynamically following any changes in the modeled attacks or in the implemented security policy. Each probe has the structure shown in Figure 3.

The *filter* module is responsible for filtering the network message stream. Its main task is to select those messages that contribute to signature actions or dynamic assertions used in a state transition scenario from among the huge number of messages transmitted over a network link. The filter module can be configured remotely by the analyzer. Its configuration can also be updated at run-time to reflect new attack scenarios, or changes in the network configuration. The performance of the filter is of paramount importance, because it has strict real-time constraints for the process of selecting the events that have to be delivered to the inference engine. In the current prototype the filter is implemented using the BSD Packet Filter [8] and a modified version of the *tcpdump* application [9].

The *inference engine* is the actual intrusion detecting system. This module is initialized by the analyzer with a set of state transition information representing attack scenarios (or parts thereof). These attack scenarios are codified in a structure called the inference engine table. At any point during the probe execution, this table consists of snapshots of penetration scenario instances (instantiations), which are not yet completed. Each entry contains information about the history of the instantiation, such as the address and services involved, the time of the attack, and so on. On the basis of the current active attacks, the event stream provided by the filter is interpreted looking for further evidence of an occurring attack. Evolution of the inference engine state is monitored by the *decision engine*, which is responsible for taking actions based on the outcomes of the inference engine analysis. Some possible actions include informing the Network Security Officer of successful or failed intrusion attempts, alerting the Network Security Officer during the first phases of particularly critical scenarios, suggesting possible actions that can preempt a state transition leading to a compromised state, or playing an active role in protecting the network (e.g., by injecting modified datagrams that reset

[2]This original formulation of the NetSTAT state transition language was subsequently refined into a general-purpose state transition language, called STATL [2].

network connections.)

Probes are autonomous intrusion detection components. If a single probe is able to detect all the steps involved in an attack then the probe does not need to interact with any other probe or with the analyzer. Interaction is needed whenever different parts of an intrusion can be detected only by probes monitoring different subparts of the network. In this case, it is the analyzer's task to decompose an intrusion scenario into sub-scenarios such that each can be detected by a single probe. The decision engine procedures associated with these scenarios are configured so that when part of a scenario is detected, an event is sent to the probes that are in charge of detecting the other parts of the overall attack. This simple form of forward chaining allows one to detect attacks that involve different (possibly distant) sub-networks.

2.1.4 Analyzer

The analyzer is used to analyze and instrument a network for the detection of a number of selected attacks. It takes as input the network fact base and a state transition scenario database and determines:

- which events have to be monitored; only the events that are relevant to the modeled intrusions must be detected;
- where the events need to be monitored;
- what information about the topology of the network is required to perform detection;
- what information must be maintained about the state of the network in order to be able to verify state assertions.

Thus, the analyzer component acts as a probe generator that customizes a number of general-purpose probes using an automated process based on a formal description of the network to be protected and of the attacks to be detected. This information takes the form of a set of probe configurations. Each probe configuration specifies the positioning of a probe, the set of events to be monitored, and a description of the intrusions that the probe should detect. These intrusion scenarios are customized for the particular sub-network the probe is monitoring, which focuses the scanning and reduces the overhead.

The analyzer is composed of several modules (see Figure 4). The network fact base and the state transition scenario database components are used as internal modules for the selection and presentation of a particular network and a selected set of state transition scenarios. The *analysis engine* uses the data contained in the network fact base and the state transition scenario database to customize the selected attacks for the particular network under exam. For example, if one scenario describes an attack that exploits the trust relationship between a server and a client, that scenario will be customized for every client/server pair that satisfies the specified trust relationship[3]. This customization allows one to instantiate an attack in a particular context. Using the description of the topology of that context it is then possible to identify what the sufficient conditions for detection are or if a particular attack simply cannot be detected given the current network configuration.

[3]Thus, state assertions are treated as if they were universally quantified.

Once the attack scenarios contained in the state transition scenario database have been customized over the given network, another module, called the *configuration builder*, translates the results of the analysis engine to produce the configurations to be sent to the different probes. Each configuration contains a filter configuration, a set of state transition information, and the corresponding decision tables to customize the probe's decision engine.

2.2 Example

Consider, as an example, an active UDP spoofing attack. In this scenario an attacker tries to access a UDP-based service exported by a server by pretending to be one of its trusted clients, that is, by sending a forged UDP-over-IP datagram that contains the IP address of one of the authorized clients as the source address. The receiver of a spoofed datagram is usually not able to detect the attack. For this example, consider the network presented in Figure 2 and assume that host `lang` is attacking host `fellini` by providing an NFS request that pretends to come from `wood`, who is a trusted, authorized client. Host `fellini` receives the request encapsulated in a link-level message from `chaplin`'s interface i_{3_3} to `fellini`'s interface i_4. Host `fellini` has no means to distinguish this message from the final link-level message used to deliver a legitimate request coming from `wood`. Therefore, `fellini` cannot determine if the datagram is a spoofed one. The spoofing can be detected, however, by examining the message on link L_2. In this case, since the link-level message comes from `bergman`'s interface i_{9_1} while it should come from `wood`'s interface i_7, the datagram can be recognized as spoofed. In general, if one considers a single link-level message that encapsulates a UDP-over-IP datagram, the datagram may be considered spoofed if there is no path between the interface corresponding to the datagram source address and the link-level message source interface in the network obtained by removing the link-level message source interface from the corresponding link.

This attack scenario is described in Figure 5 using a state transition diagram. The scenario assumes that two networks have been defined, `Network` and `ProtectedNetwork`. `Network` is a reference to the network modeled in the fact base; `ProtectedNetwork` is a sub-network that contains the hosts that must be protected against the attack.

The starting state (S_1) is characterized by assertions that define the hosts, interfaces, addresses, and services involved in the attack. The first assertion states that the attacked host `victim` belongs to the protected network. The second assertion states that there is a service `s` in the set of services provided by `victim` such that the transport protocol used is UDP, and service authentication is based on the IP address of the client. The third assertion states that `a_v` is one of the IP addresses where the service is available. The fourth assertion says that `a_t` is one of the addresses that the service considers as "trusted". The following assertions characterize the attacker. In particular, the fifth assertion states that there exists a host `attacker` that is different from `victim` and that doesn't have the trusted IP address. The sixth assertion states that `i` is one of the `attacker`'s interfaces.

The signature action is a spoofed service request. That is, a UDP datagram that pretends to come from one of the trusted addresses, although it did not originate from the corresponding interface. Actually the signature action is a link-level message `m` that belongs to the sequence of mes-

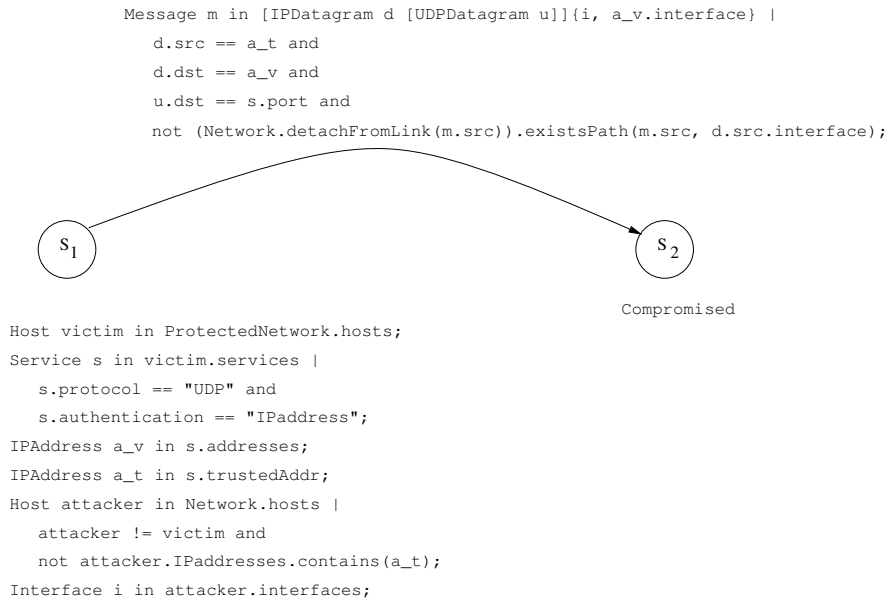


Figure 5: UDP spoofing attack scenario.

sages used to deliver an IP datagram from interface `i` to the interface associated with the address of the attacked host. The IP datagram enclosed in the message has source address `a_t` and destination address `a_v`. The IP datagram encloses a UDP datagram, whose destination port is the port used by service `s`. In addition, the message is such that, if one considers the network obtained by removing the message source interface from the corresponding link (i.e., `Network.detachFromLink(m.src)`), there is no path between the interface corresponding to the datagram IP source address and the link-level message source interface. For example, consider a link-level message exchanged between `bergman`'s interface i_{9_1} and `chaplin`'s interface i_{3_2}. The message is an intermediate step in the delivery of a UDP-over-IP datagram to `fellini`; the IP source address of the datagram is `wood`'s a_7. Intuitively, it is clear that a message originated by `wood` and intended for `fellini` cannot come from one of `bergman`'s interfaces, because there is no path in the network that would require `bergman` to act as a forwarder of the datagram. One way to check for this is by removing the source interface of the message (i_{9_1}) and checking whether or not there still exists a path from the host whose IP address is the source of the datagram (`wood`) to the host that contains the interface that was removed (`bergman`). The second state (S_2) is a "compromised" state.

The analysis of the attack starts by identifying the possible scenarios in the context of a modeled network. Thus, the analysis engine determines all the possible combinations of victim host, attacked service, spoofed address, and attacker in a particular network. A subset of the scenarios for the network in Figure 2 is presented in Table 1. In all scenarios `fellini` is the attacked host, NFS is the service exploited, and the spoofed address can be `kubrick`'s or `wood`'s.

The next step in the analysis is to determine where the events associated with the signature action can be detected. For each of these scenarios, the analysis engine generates all the possible datagrams between the interface of the attacker and the interface of the victim. In practice, the engine finds all the paths between the interfaces defined by the scenario and, for each path, generates the sequence of messages that would be used to deliver a datagram. For each message the predicate contained in the clause of the signature action is applied. The messages that satisfy the predicate are candidates for being part of the detection of the scenario. For example, consider the scenario where `carpenter` is attacking `fellini` by pretending to be `wood`. In this case, the spoofed datagram is generated from interface i_{11} and delivered through three messages to `fellini`'s interface i_4. The first message is between `carpenter` and `bergman`, the second one is between `bergman` and `chaplin`, and the third one is between `chaplin` and `fellini`. Of these three messages only the first two satisfy the predicate of the signature action. Therefore, to detect this particular scenario one either needs a probe on L_3 looking for link-level messages from `carpenter`'s interface i_{11} to `bergman`'s interface i_{9_2}, or a probe on L_2 looking for messages from `bergman`'s interface i_{9_1} to `chaplin`'s interface i_{3_2}. In both cases, the IP source address is a_7, the destination IP address is a_4, and the destination UDP port is the one used by the NFS service. By analyzing all the scenarios, one finds that in order to detect all possible spoofing attacks it is necessary to set up probes on links L_1, L_2, and L_4.

3. EVALUATING INTRUSION DETECTION SYSTEMS

Network intrusion detection systems should not be difficult to evaluate: given a traffic dump collected during real or simulated intrusions, a NIDS should be able to detect a subset of the attacks while producing a certain (hopefully low) number of false positives. This is not as straightforward with other types of intrusion detection systems (e.g., host-based systems and application-based systems), because the quantity and quality of information collected about the actions performed by the OS and its applications can vary dramatically. In addition, systems that use an anomaly-

victim	s	a_v	a_t	attacker	i
fellini	NFS	a_4	a_5	Outside	i_0
fellini	NFS	a_4	a_7	Outside	i_0
fellini	NFS	a_4	a_5	hitchcock	i_{1_1}
fellini	NFS	a_4	a_7	hitchcock	i_{1_1}
...	...	...	...	...	...
fellini	NFS	a_4	a_5	lang	i_{10}
fellini	NFS	a_4	a_7	lang	i_{10}
fellini	NFS	a_4	a_5	carpenter	i_{11}
fellini	NFS	a_4	a_7	carpenter	i_{11}

Table 1: Possible scenarios for the UDP spoofing attack.

based approach to intrusion detection necessitate training data, which should be realistic, complete, and attack-free (note that "realistic and attack-free" could be considered an oxymoron). This type of data is particularly hard to collect and/or generate.

Even though the creation of a dataset that can be leveraged to compare the performance of intrusion detection systems is a challenging task, in 1998 and 1999 a group of researchers from the MIT Lincoln Laboratory courageously embarked in an effort to produce such a dataset, which included both training data and test data (with truth files) in the form of network packets, OS audit records, and file system dumps [6, 7].

These datasets were used to evaluate a number of intrusion detection systems being developed by academic researchers. At the beginning of the evaluation process, the attack-free training data was given to the participants, and, after a while, the test data containing attacks was distributed (without truth files). The participants then had to identify the attacks and submit their detection alerts, which were then evaluated with respect to the truth files.

The results of the evaluation were disclosed only partially, without declaring a "winner," and with great care in not making any single group look bad. Therefore, instead of a single score, the authors of the evaluation provided a set of scores that took into consideration various characteristics of the systems involved, creating a no-winner/no-loser situation. We think that this was a missed opportunity to foster research by creating a competition with a clear winner, as was later demonstrated by other challenges (e.g., the DARPA Grand Challenge for unmanned vehicles), which by having a clear winner motivated the competitors, fostered innovation and creativity, and provided great publicity for both the participants and the funding agency.

To determine a winner, a more draconian approach to evaluation would have been to simply compose the recall and precision of the intrusion detection systems. More precisely, in order to evaluate the effectiveness of a system one could compute the percentage of hits H over the total number of attacks T, that is, $(H/T) * 100$. This is a measure of how many attacks were actually detected with respect to the overall set of attacks (i.e., the recall). Then, in order to characterize how precise the system is, one would compute the percentage of false alarms F over the total number of detections $H + F$, that is $(H/(H + F)) * 100$. For example, a system with three detections and no false alarms would have a precision of 100%, but it would not be very effective at detecting attacks if the dataset contained hundreds of attack instances. As another example, a system that flagged every single packet as malicious would have an effectiveness of 100% because all attacks would be detected, but it would also have an abysmal precision. Therefore, the obvious choice is to multiply the two measures above to take into account these two important aspects of intrusion detection.

The values of these metrics are shown in Table 2 for the systems that participated in the 1999 MIT Lincoln Laboratory evaluation. According to the proposed metrics, UCSB's NetSTAT would be the winner of the 1999 competition, closely followed by SRI's EMERALD.

Even though the evaluation failed to declare a clear winner and, in addition, there were some criticisms against the evaluation process [10], the dataset produced was immensely popular, and it is without doubt the most used dataset in the intrusion detection community.

Unfortunately, the MIT/LL dataset and the corresponding truth files were used in a series of scientific publications in which the performance of intrusion detection systems, evaluated on the non-blind dataset, were compared to the performance of the intrusion detection systems that participated in the blind evaluation, with nefarious and unfair results. Since then, the dataset has become outdated, and nowadays it is used very seldom in research publications.

4. THE DEATH (AND REBIRTH) OF INTRUSION DETECTION

In the years following the MIT/LL evaluation, there was an increased skepticism towards network intrusion detection and its ability to detect attacks, especially 0-day exploits and mutations of existing attacks [17]. In addition, researchers started to develop attacks against stateful intrusion detection system, exposing the challenge of detecting low-traffic, slow-paced attacks that last months (if not years).

In general, there was a shift from the analysis of network data to the analysis of host data, under the assumption that only by monitoring the end nodes one could possibly detect sophisticated attacks. Therefore, during the early 2000s, academia started losing interest in network intrusion detection, while, at the same time, the use of commercial network intrusion detection systems became an established best-practice in enterprise networks. This happened sometimes in disguise, for example by relabeling NIDS as "intrusion prevention systems" to describe network intrusion detection systems with traffic-blocking responses.

Around 2003-2004 it looked like research on the "classic"

	GMU	NYU	RST Elman Network	RST State Tester	RST String Transd.	SRI Derbi	SRI Estat	SRI EMERALD	SunySB	UCSB STAT
Hits	43	21	37	26	26	17	29	94	7	88
False Positives	16	74	5351	429	117	48	96	13	2	4
H/T	21.3	10.4	18.3	12.9	12.9	8.4	14.4	46.5	3.5	43.6
$H/H+F$	72.9	22.1	0.7	5.7	18.2	26.2	23.2	87.9	77.8	95.7
$H/T * H/H+F$	15.5	2.3	0.1	0.7	2.3	2.2	3.3	40.9	2.7	41.7

Table 2: Hits, false positives (in absolute values), recall, precision, and composition of recall and precision (in percentages) for the systems involved in the MIT Lincoln Laboratory 1999 IDS evaluation, which contained 202 attacks.

network intrusion detection problem (i.e., detecting attacks by looking at network packets) was dwindling fast. However, at the same time, the techniques used to characterize network attacks were applied to the detection of malicious code components, such as worms and bots. Both misuse-based and anomaly-based techniques were readily leveraged to identify malware of various kinds. In a way, these research efforts resulted in "intrusion detection" system that were closer to the meaning of the term than the early NIDSs. In fact, while the early systems focused mostly on detecting attacks, these new systems focused on detecting the actual intrusions by identifying malicious behavior that would indicate that a system had been compromised.

This "born-again" network intrusion detection research is characterized by the heavy use of data-mining and machine-learning techniques to address one of the main problems associated with misuse-based NIDS, which is the need for the manual specification of attack models (note that some of the seminal work in this field was performed in the late 90's [5]).

5. CONCLUSIONS

Even though the term "Intrusion Detection" sometimes is looked-down upon by the academic community, intrusion detection research will always be a core part of the security field. It might be the case that the focus of intrusion detection will move towards more semantically-rich domains, such as the OS and the web. For example, web-based intrusion detection systems (normally referred to as "Web Application Firewalls", for marketing purposes) leverage knowledge about the characteristics of web applications and their logic, in order to identify attacks. Nonetheless, these systems mostly use concepts that were researched and applied more than two decades ago.

This "re-invention" of network intrusion detection techniques and approaches shows how intrusion detection (be it network-based, web-based, or host-based) is still an important research problem. As new attacks and new ways of compromising systems are introduced, both researchers and practitioners will develop (or re-discover) techniques for the analysis of events that allow for the identification of the manifestation of malicious activity.

The next challenge will be to expand the scope of intrusion detection to take into account the surrounding context, in terms of abstract and difficult-to-define concepts, such as missions, tasks, and stakeholders, when analyzing data in an effort to identify malicious intent.

6. REFERENCES

[1] C. Berge. *Hypergraphs*. North-Holland, 1989.

[2] S. Eckmann, G. Vigna, and R. Kemmerer. STATL: An Attack Language for State-based Intrusion Detection. *Journal of Computer Security*, 10(1,2):71–104, 2002.

[3] L. Heberlein, G. Dias, K. Levitt, B. Mukherjee, J. Wood, and D. Wolber. A Network Security Monitor. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 296 – 304, Oakland, CA, May 1990.

[4] K. Ilgun, R. Kemmerer, and P. Porras. State Transition Analysis: A Rule-Based Intrusion Detection System. *IEEE Transactions on Software Engineering*, 21(3):181–199, March 1995.

[5] W. Lee and S. Stolfo. Data Mining Approaches for Intrusion Detection. In *Proceedings of the USENIX Security Symposium*, San Antonio, TX, January 1998.

[6] R. Lippmann, D. Fried, I. Graf, J. Haines, K. Kendall, D. McClung, D. Weber, S. Webster, D. Wyschogrod, R. Cunningham, and M. Zissman. Evaluating Intrustion Detection Systems: The 1998 DARPA Off-line Intrusion Detection Evaluation. In *Proceedings of the DARPA Information Survivability Conference and Exposition, Volume 2*, Hilton Head, SC, January 2000.

[7] R. Lippmann and J. Haines. Analysis and Results of the 1999 DARPA Off-Line Intrusion Detection Evaluation. In *Proceedings of the Symposium on the Recent Advances in Intrusion Detection (RAID)*, pages 162–182, Toulouse, France, 2000.

[8] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the 1993 Winter USENIX Conference*, San Diego, CA, January 1993.

[9] S. McCanne, C. Leres, and V. Jacobson. Tcpdump 3.7. Documentation, 2002.

[10] J. McHugh. Testing Intrusion Detection Systems: A Critique of the 1998 and 1999 DARPA Intrusion Detection System Evaluations as Performed by Lincoln Laboratory. *ACM Transaction on Information and System Security*, 3(4), November 2000.

[11] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, January 1998.

[12] P. Porras. STAT – A State Transition Analysis Tool for Intrusion Detection. Master's thesis, Computer Science Department, University of California, Santa

Barbara, June 1992.

[13] P. Porras and P. Neumann. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In *Proceedings of the 1997 National Information Systems Security Conference*, October 1997.

[14] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the USENIX LISA '99 Conference*, Seattle, WA, November 1999.

[15] G. Vigna. A Topological Characterization of TCP/IP Security. In *Proceedings of the* 12^{th} *International Symposium of Formal Methods Europe (FME '03)*, number 2805 in LNCS, pages 914–940, Pisa, Italy, September 2003. Springer-Verlag.

[16] G. Vigna and R. Kemmerer. NetSTAT: A Network-based Intrusion Detection Approach. In *Proceedings of the* 14^{th} *Annual Computer Security Applications Conference (ACSAC '98)*, pages 25–34, Scottsdale, AZ, December 1998. IEEE Press.

[17] G. Vigna, W. Robertson, and D. Balzarotti. Testing Network-based Intrusion Detection Signatures Using Mutant Exploits. In *Proceedings of the ACM Conference on Computer and Communication Security (ACM CCS)*, pages 21–30, Washington, DC, October 2004.

Barriers to Science in Security

Tom Longstaff
The Johns Hopkins University
Applied Physics Laboratory 11100
Johns Hopkins Rd., Laurel, MD
20723
Thomas.Longstaff@jhuapl.edu

David Balenson
The Johns Hopkins University
Applied Physics Laboratory 11100
Johns Hopkins Rd., Laurel, MD
20723
Thomas.Longstaff@jhuapl.edu

Mark Matties
The Johns Hopkins University
Applied Physics Laboratory 11100
Johns Hopkins Rd., Laurel, MD
20723
Thomas.Longstaff@jhuapl.edu

Overview

In the past year, there has been significant interest in promoting the idea of applying scientific principles to information security. The main point made by information security professionals who brief at conferences seems to be that our field of information security is finally mature enough to begin making significant strides towards applying the scientific approach. Audiences everywhere enthusiastically agree and thrash themselves for bypassing science all along, bemoaning the fact that we could be "so much further along" if we only did science. Of course, after the presentation is over, everyone goes back to the methods that have been used throughout our generation to generate prototypes and tools with no regard for the scientific principles involved.

The type of information security[1] projects in scope for this essay are experimental projects that produce a new approach or support/refute a theoretical result. The use of the scientific method in theoretical information security and in computer science more generally is well documented and mature (even if not universally applied). The focus of the "science of security" publications in FY09-10 is in the area of experimentation and applied information security research. Thus our focus here is also in the comparison of experimental information security research that does or does not use a traditional scientific method in the execution of the project and in the publication of the results. The definition of the scientific method we use in this essay is well documented and not further described here.

Finding agreement in the use of the scientific method is practically universal, finding participation in the scientific method is rare. Why? What are the primary barriers to applying the scientific method to information security projects? What are the main differences between the projects that apply the scientific method to experimental information security projects and those that promote software/tool development without applying a traditional scientific approach? In this essay, I explore three main barriers to achieving a more universal application of the scientific method to experimental information security projects. These are:

- Time to publish as a primary driver
- Standard of peer reviews in conferences and journals
- Expectation of a breakthrough in every publication

Although these drivers are evident in many academic publications, it must be noted that much of the work in computer science, and more importantly, information security does not concern the development of the body of scientific knowledge, but in getting a job done. This is closely aligned to computer engineering or software engineering, both of which are often associated with computer science departments. Many practicing computer scientists work in the area of information security by producing innovative tools and techniques to solve specific technical problems in information security. Many of these practitioners have a computer science degree, but have never been formally trained in the application of scientific method, nor do they need to be to have successful careers in information security. The overarching goal in this area of information security is to get the job done in terms of writing a program to accomplish a task, rather than on exploring the and testing the range of possibilities (experimenting) and implementing a better solution. Practical, working systems that can be quickly implemented tend to prevail. This follows the old IETF mantra of "rough consensus and running code." (see http://en.wikipedia.org/wiki/Rough_consensus).

In this sense, the Science in "Computer Science" is a misnomer – many CS graduates are never formally trained in the scientific method and its use in experimental information security. Some CS curricula teach basics in terms of computational logic, programming languages, data structures, database, artificial intelligence, etc., but do not teach scientific experimentation. Many other academic curricula, like Math, and even English, often develop students who ultimately work as programmers, developers, or researchers, but they also lack formal education in scientific method. The curricula that do teach scientific methods, such as Psychology, Biology, Physics, etc. lead to few people who work on information security. However, as more of these professionals enter the field, the call for a scientific approach becomes increasingly urgent.

[1] We use the term *information security* to clarify that the types of projects in scope address the confidentiality, integrity, or availability of information assets. While it is common to use the term *cyber security* to address perhaps a wider set of topics, the definition of *cyber security* is not as well defined or accepted, and thus is more likely to cause confusion over the types of projects included herein.

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

Time to Publish as a Primary Driver

The application of the scientific method to experimental information security projects usually takes significantly more time than is available for the development of a demonstration/prototype tool. A carefully conceived experiment requires planning around a well-formed hypothesis, assuring that the tests against the hypothesis are sufficient to potentially refute the hypothesis. In the likely event that the experiment will support the hypothesis, the domain of the test environment must be sufficient to build an argument that the hypothesis holds in a significantly extensive context. This frequently means many runs of the experiment over a wide variety of input variations to assure the relationship between the domain and range of the system under test (SUT) is as predicted by the hypothesis.

In contrast, many experimental tests take a developed prototype or demonstration system and provides a narrow set of performance characteristics. Since there is no hypothesis to test, there is no possibility of refuting a hypothesis. All that is generated is a series of observations of the SUT. The tests can be performed in a narrow set of domain variables since the test is designed to show performance in the environment for which the SUT was designed. Since no failure is possible in this situation, the tests need not be extensive to lead to results that may be published.

Even when a rigorous scientific test is designed, the pressure to publish quickly may lead to an inadequate exploration through extensive and multiple trials. There is a tendency to test a very limited set of functionality or a small number of parameters. This approach supports the hypothesis, but only for a limited environment. These tests answer specific questions such as testing an implementation X in environment Y and it's ability to detect Z. Variations X' in alternative environments Y' may be limited. The full operating range or characteristics of our technology may not be included in the rush to publish.

The publication of a well-designed experiment must follow a rigorous structure that will allow readers of the publication to fully repeat the experiment. This includes the domain (data and input settings), full description of the SUT (including any implementations), and the architecture of the test environment. This implies that this data was carefully captured during the experiment, which again takes a carefully planned experimental methodology. When simply executing performance tests of a prototype/demonstration system, the standard is not to capture the experiment in full detail, but to instead describe the performance of the prototype/demonstration. The publication is not designed to allow others to re-create the experiment but instead to motivate the use of the prototype in their environment. Since there is not carefully described domain description for the test, the result (range) of the prototype in a new environment cannot be accurately predicted.

A well-defined experiment has a much more powerful predictive value, but given that it takes a much longer time to achieve, there is significant pressure on researchers to publish a higher volume of results more quickly than running a series of experiments. Since the metric for most academics in the area of information security is number of publications rather than quality of experimental results, rewards are gained by minimizing a scientific approach and putting out as many publications on new prototypes/demonstrations as possible. Since we get what we incentivize, time to publish becomes a primary driver for choosing prototyping over science.

Standard of Peer Reviews in Conferences and Journals

Of course, rapidly producing many publication submissions based on prototypes and demonstrations would be irrelevant if the selection criteria in conferences and journals favored science over demonstration.

In many natural and social science journals and conferences, a submission must demonstrate the use of good science principles in order to be considered for publication. In scientific areas such as Physics, Chemistry, Psychology, and many others, the entire culture is focused on the critical evaluation of scientific evidence. A reviewer in these disciplines has an enormous responsibility to represent the critical review of the entire readership. Her primary responsibility is to discredit the potential publication before it can be discredited by the readership. A well respected journal or conference gains a reputation for the inclusion of only a small subset of submissions that cannot be discredited, so thus must be published to allow another researcher to reproduce the results or possibly refute the hypothesis while hopefully proposing an alternative.

In cultures such as the natural and social sciences described above, critical reviewers are trained throughout their career to evaluate submissions for scientific rigor. New ideas are not simply given credence for being clever, but must be supported with scientific evidence. Only then can the new idea be incorporated into the scientific body of knowledge and used to make further predictions.

In sharp contrast to this culture, the majority of the information security reviewers consider the technological implementation of new ideas to be of high worth. A description of a new tool that implements a feature that has not yet been conceived is of great interest to most of the reviewing community. A critical review of this type of submission usually focuses on the quality of the description itself, and of any duplication the tool might have with previous tools that have been created (often to assure there is a reference to this prior work). In this case, experimental design is neither desired nor appreciated in the submission, and may be excluded for a reduced page count.

Expectation of a Breakthrough in Every Publication

If you accept the previous two points (time to publish and standard of peer review) as driving the culture of scientific discourse in information security, a natural expectation for short-term R&D is to create a novel new system and publish the result. These new systems are designed to solve particular problems (such as intrusion detection or secure computing), but the approach to solving the problem is to use insight to create a novel solution that attempts to solve the problem at large. The "breakthrough" solutions are shown to be effective in a lab environment or small set of enterprise environments and described as a prototype demonstration of the novel concept.

While there is absolutely nothing wrong with the generation of technology based on novel concepts (this is how many companies succeed), this is not a scientific approach to solving problems in information security. Using a scientific approach would create reusable knowledge or explore causal relationships rather than

focus on the apparatus used to gain these results. By equating the process of "scientific discovery" with technology innovation, we create an expectation that scientific publications should always contain a breakthrough technology as a core benefit. This expectation leads to a reduced number of accepted publications that show incremental progress in the understanding of how information security actually works, and instead promotes publications that fully describe a technology breakthrough.

Conclusions and Way Forward

It is certainly possible that in this field, the traditional scientific approach is not commercially viable from a product development standpoint. It can easily be argued that given the rapid pace of technological advance, we should be promoting innovative technological solutions over scientific investigation. We do have mature and rigorous scientific investigation in computer science more generally and in information security from a theoretical and cryptographic perspective. While we don't often use these results to drive innovation, there are specific instances where we have used results from theoretical computer security to drive a security product.

If this is the case, why the clamor for scientific method in experimental information security? Given the advances in other experimental sciences, the hope is that we can begin to develop lines of information security products that are incrementally better as time goes on, not just by adding features to an implementation, but by understanding the underlying causality of information security and addressing the problem at its most fundamental level. Applying the scientific method to our experiments will enable a more purposeful approach to discovering the exact conditions under which our innovations can be expected to operate, providing much greater utility in our future products.

If this is a goal to be at least partially achieved, the three barriers to adoption described in this article must be addressed. Each of these poses a significant challenge to the field as they address the culture of our process, which one can argue has successfully produced commercially successful products. Yet the basic problem of information security remains. Could we begin to eliminate these problems through the application of experimental science in information security? If we do not create at least a small sub-culture that applies scientific method to experimental information security, we may never know. If we do create such a sub-culture that embraces experimental science in information security, it might be best to treat this delicate new community as a "skunk-works" from the main body of information security R&D. This would involve creating a series of publication venues that use reviewers from this new community, create expectations that will appeal largely to this community (and not to the information security community at large), and which creates a body of knowledge that is formed outside of the mainstream of information security R&D. The success or failure of this community will pivot on its ability to solve fundamental questions in information security in a way that cannot be ignored by the mainstream.

It possible that the current climate of our funding agencies in the US and EU are disposed to fund the creation of this community given a clear definition and leadership in its formation. For members of this conference that both have a deep understanding and appreciation of experimental science and for future program managers that might fund such an approach, it is time to come together to produce the "grand experiment" of the creation of a sub-community of information security that rejects ad-hoc solutions in favor of scientific evidence that increase our understanding of information security.

Friends of An Enemy: Identifying Local Members of Peer-to-Peer Botnets Using Mutual Contacts

Baris Coskun*
AT&T
33 Thomas Street
New York, NY
baris@att.com

Sven Dietrich
Stevens Inst. of Technology
Castle Point on Hudson
Hoboken, NJ
spock@cs.stevens.edu

Nasir Memon
Polytechnic Institute of NYU
Six Metrotech Center
Brooklyn, NY
memon@nyu.edu

ABSTRACT

In this work we show that once a single peer-to-peer (P2P) bot is detected in a network, it may be possible to efficiently identify other members of the same botnet in the same network even before they exhibit any overtly malicious behavior. Detection is based on an analysis of connections made by the hosts in the network. It turns out that if bots select their peers randomly and independently (i.e. unstructured topology), any given pair of P2P bots in a network communicate with at least one mutual peer outside the network with a surprisingly high probability. This, along with the low probability of any other host communicating with this mutual peer, allows us to link local nodes within a P2P botnet together. We propose a simple method to identify potential members of an unstructured P2P botnet in a network starting from a known peer. We formulate the problem as a graph problem and mathematically analyze a solution using an iterative algorithm. The proposed scheme is simple and requires only flow records captured at network borders. We analyze the efficacy of the proposed scheme using real botnet data, including data obtained from both observing and crawling the Nugache botnet.

Categories and Subject Descriptors

C.2.0 [**Computer Communication Networks**]: General—*Security and protection (e.g., firewalls)*; C.2.3 [**Computer Communication Networks**]: Network Operations—*Network Monitoring*

General Terms

Security

Keywords

P2P Botnet, IDS, Network Security

*This work was carried out while Baris Coskun was with Polytechnic Institute of NYU

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

1. INTRODUCTION

Botnets, which are networks of compromised hosts (bots) under the control of a botmaster, have become a major threat in recent years. Botnets are used to perform various malicious activities such as spamming, phishing, stealing sensitive information, conducting distributed denial of service (DDoS) attacks, scanning to find more hosts to compromise, etc. Bots which perform such malicious activity, occasionally go over the radar and get detected by Intrusion/Anomaly/Behavior Detection Systems present within the network. In fact, network administrators routinely discover bots which are then immediately quarantined or removed. However, some interesting and important questions remain, such as: ***"Does the network contain more bots of the same type which haven't been exposed yet?" "Can the discovered bot be leveraged to find other dormant bots of the botnet before they commit any malicious activity?" "Can all this be done without any access to backbone traffic and only from netflow data from the edge router?"***

A common and fairly obvious approach to find dormant bots is to characterize the Command and Control (C&C) channel from the discovered bot's recent traffic and identify hosts that exhibit similar C&C traffic characteristics. For example, botnets with a centralized C&C architecture, where all bots receive commands from a few central control servers, the source of the C&C messages can be used to characterize the corresponding C&C channel and reveal potential dormant bots [23].

However, characterizing the C&C channel is in general not a trivial task for botnets that utilize a *peer-to-peer* (P2P) architecture involving no central server. For example, this kind of source analysis falls short for P2P botnets as here the botmaster can use any node to inject C&C messages. To receive and distribute C&C messages, each P2P bot communicates with a small subset of the botnet (i.e. peer list) [30, 14, 18] and maintains its own peer list independently. Hence, no obvious common source of C&C messages can be observed, thereby preventing the linking of the discovered bot with the dormant bots. Furthermore, features based on packet sizes and timings, such as packets per flow, bytes per flow, flows per hour, etc. may not be useful in characterizing a C&C channel, since botmasters can easily randomize such features thereby obtaining different feature values for each bot [29]. Botnets such as Nugache, Storm, Waledac and Conficker employ advanced encryption mechanisms [30, 14, 18, 28, 27] making characterization based on packet contents

infeasible.

In this paper we propose an efficient technique to discover additional P2P bots in a network after one such bot has been discovered. Specifically, the proposed technique provides a list of hosts ordered by a degree of certainty, that potentially belong to the same P2P botnet as the discovered bot. Network administrators can use this list as a starting point of their investigation and potentially identify more bots in their network once they discover one. The proposed technique is based on the simple observation that peers of a P2P botnet communicate with other peers in order to receive commands and updates. Although different bots may communicate with different peers, we show that for P2P botnets with an unstructured topology, where bots randomly pick peers to communicate with, there is a surprisingly high probability that any given pair of P2P bots communicate with at least one common external bot during a given time window. In other words, there is a significant probability a pair of bots within a network have a **mutual contact**. We present a precise mathematical derivation of this probability as a function of the size of a botnet and the number of peers a bot contacts. Notice that we focus on P2P botnets with unstructured topology in this work and the term "P2P botnet" refers to unstructured P2P botnets in the rest of the paper unless stated otherwise.

In order to discover dormant bots, we first construct a mutual contacts graph where every host is a node and two nodes share an edge if they share a mutual contact. The weight or capacity on an edge is the number of mutual contacts shared between the corresponding hosts incident on the edge. Then given a discovered bot or seed bot, we present an iterative algorithm, which identifies other potential members of the botnet by iteratively computing a level of confidence to each host on the graph. We declare the hosts which have confidence levels higher than a threshold as potential members of the same P2P botnet as the seed-bot. We present experimental results with real and simulated traffic to measure the effectiveness of our technique. We also present mathematical analysis characterizing the structure of a mutual contact graph.

In addition to being simple and effective the proposed scheme has the following desirable properties:

- The proposed method is not an anomaly detection scheme and hence doesn't require P2P bots to exhibit any overtly malicious activity.
- Similarly, it is not a behavior clustering algorithm and therefore doesn't require any common behavior exhibited by all the bots.
- It utilizes the pairwise mutual-contact relationships between pairs of bot peers, which arise due to P2P C&C communications. We validate the existence of such relationships both mathematically and experimentally.
- The proposed method is generic and doesn't depend on specific properties of specific botnets. Therefore, it doesn't require reverse engineering bot binaries or C&C protocols [3].
- Contrary to existing graph-based network traffic analysis methods [26] [19], the proposed method doesn't require any access to backbone traffic. Mutual-contact relationships are deduced locally at an edge router.

In the rest of this paper, we explain the basic idea and details of the proposed method in Section 2. Following that, we present our experimental results with the Nugache botnet in Section 3. In Section 4, we present a mathematical analysis that provides insights on why the proposed scheme works and its limitations. Then in Section 5, we discuss practical limitations of the proposed scheme, possible evasion techniques and their implications on P2P botnets. We present the related work in Section 6. Finally, we conclude the paper and discuss future work in Section 7.

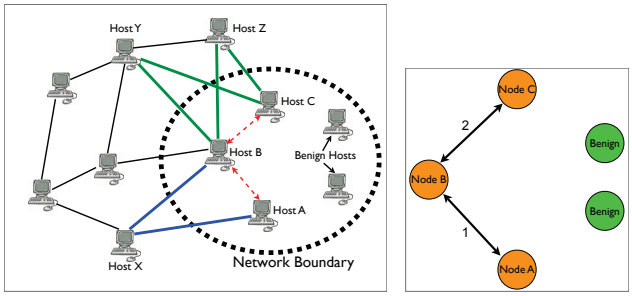


Figure 1: Illustration of P2P Botnet communications for a network (a) and its corresponding mutual-contact graph (b). The network contains 2 benign hosts and 3 bots (Hosts A, B, and C). The bots are members of a P2P botnet with 9 bots in total. Mutual-contact relationship among hosts, which is indicated by red dashed arrows in (a), are represented by the mutual-contact graph in (b). The edge capacities are determined by the number of mutual contacts between nodes.

2. FINDING FRIENDS OF AN ENEMY

In this section, we present the basic idea and the details of the proposed algorithm. We first begin with an intuitive explanation in the next subsection and then provide a more detailed and formal explanation in subsequent subsections.

2.1 Basic Idea

Consider the botnet illustrated in Figure 1(a). The basic idea of the proposed method is that, *Host A* can be linked to *Host B* since they both communicate with *Host X* (the mutual contact). Similarly *Host B* and *Host C* are linked together through *Host Y* and *Host Z*. As a result, if *Host A* becomes known as a member of a P2P botnet, then by examining its connections, one may suspect that *Host B* is likely to be a member due to the presence of a mutual contact with the known bot *Host A*. Similarly, if *Host B* is likely to be a member, then *Host C* is likely to be a member as well.

Now it is clear that, aside from P2P botnet traffic, legitimate traffic probably includes several mutual-contacts among hosts as well. For instance, there are some very popular servers that almost every host in the network communicates with such as *google.com*, *microsoft.com*. etc. As a result, every host in the network would be linked to most of the other hosts through such popular mutual-contacts. However, if *H*ost A is a known bot and both *H*ost A and *H*ost B have been in communication with *H*ost X, and *H*ost X has not talked to almost anyone else within our network, then it is likely that *H*ost B is a member of the same botnet as *H*ost A. Hence in our mutual contact based analysis we restrict ourselves to **private mutual-contacts**. Private mutual contacts are mutual contacts which communicate with less than k internal hosts during an observation win-

dow. Here, k is the **privacy-threshold**. Private mutual contacts capture the intuition that it is very unlikely that external peers which are part of a botnet will be communicating with many internal hosts that do not belong to the botnet. Therefore, private mutual-contacts can be strong indicators of peer relationships among hosts within a botnet. **In the rest of this paper, we use the term mutual-contacts to mean private mutual contacts.**

The question then remains that given a known bot, how do we systematically rank all the hosts in our network based on their likelihood of being a member of the same P2P botnet using private mutual contact relationships they exhibit? To do this, we first extract mutual-contacts from the flow records captured at the network border for a time window prior to discovering the seed-bot. We then represent the mutual-contact relationships among hosts by a directed graph called the *mutual-contacts graph*, such that: 1. *Nodes represent the hosts in the network.* 2. *There is a bidirectional edge between two nodes if the corresponding hosts have at least one mutual-contact during the given time window.* 3. *Each edge has a capacity determined by the number of mutual-contacts between corresponding nodes.*

As an example, the mutual-contact graph for the network illustrated in Figure 1(a) is shown in Figure 1(b). Now intuitively speaking, it is expected that hosts which are likely to be P2P bots are at a short distance from the seed-bot on a mutual-contacts graph since such hosts are expected to have mutual-contacts with the seed-bot itself and/or with other hosts which have mutual-contacts with the seed-bot. In fact, we observe this behavior in various real world botnets as presented later in Table 1. Furthermore, the more the mutual contacts that a host has with the seed bot and other suspected bots, the more likely it is that this host is also a bot. The mutual contacts graph illustrated in Figure 2(a) displays such behavior (black edges). Based on these two intuitions, we propose a scheme that iteratively computes a confidence level of being a member of the same P2P botnet as the seed bot for each node. This iterative process can be illustrated as pumping red dye into the mutual-contacts graph from the node representing the seed-bot as depicted in Figure 2(b). During the process, the dye coming to a node is distributed across its outgoing edges proportional to their capacities. Therefore, the dye accumulated in a node reflects our confidence for that host being a part of the same botnet as the seed-bot. Inspired by this illustration, we named our proposed algorithm the "Dye-Pumping Algorithm".

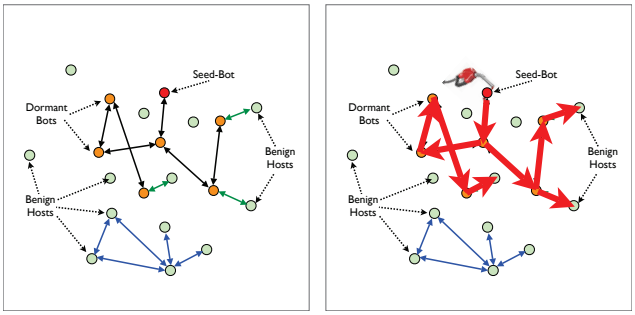
In Figure 2(b), it is also observed that along with the P2P bots, few benign hosts also share mutual-contacts with P2P bots (via green edges in Figure 2(a)), and therefore receive some amount of dye. Such hosts potentially result in false positives. However, we expect the capacity of the edges connecting these benign hosts to P2P bots to be usually lower thereby keeping the dye accumulated on these benign hosts below a threshold in most cases. In later sections we provide detailed experimental and mathematical analysis, that supports our intuition that a majority of the false positives can be eliminated while maintaining reasonable false negatives, by choosing a suitable threshold. But first, in the following subsections, we present each step of this algorithm in greater detail.

2.2 The "Mutual-Contacts" Graph

We denote the mutual-contacts graph constructed from the flow records of a network by $G = (N, E)$, where the nodes correspond to hosts and the edges indicate private mutual-contacts. Each edge on the graph has a capacity which is determined by the exact number of mutual-contacts between corresponding hosts. More formally, if E_{ij} represents the capacity of the edge between nodes N_i and N_j, then we can write:

$$E_{ij} = E_{ji} = |S(N_i) \cap S(N_j)|$$

where $S(N_i)$ represents the set of mutual-contacts which N_i was in communication with during the observation period and $|\cdot|$ represents the cardinality of a set. Notice that, if nodes N_i and N_j don't share any mutual-contacts then there is no edge between them on the graph or equivalently $E_{ij} = 0$.



(a) (b)

Figure 2: (a) Illustration of a mutual-contacts graph. P2P bots tend to share mutual-contacts with each other (black edges). Also some benign hosts share mutual-contacts among themselves (blue edges), which may be due to a legitimate P2P application. (b) Illustration of the dye-flow in the Dye-Pumping algorithm.

2.3 The "Dye-Pumping" Algorithm

Once the mutual-contacts graph is constructed, the dye-pumping algorithm is executed to compute the confidence levels of hosts being part of the P2P botnet. The dye-pumping algorithm iteratively pumps dye to the mutual-contacts graph from the seed node and picks the nodes which accumulates more dye than a threshold. During the process, dye coming to a node is distributed to other nodes proportional to a heuristic called the **dye-attraction coefficient**, which helps the algorithm to funnel more dye towards the nodes which are more likely to be P2P bots.

Dye-Attraction Coefficient is denoted by γ_{ji}, and indicates what portion of the dye arriving at Node j will be distributed to Node i in the next iteration. Intuitively, it represents our confidence on Node i being a P2P bot given that Node j is a P2P bot. Such confidence gets higher as Node i and Node j share more private mutual-contacts with each other. On the other hand, our confidence reduces if Node i shares mutual-contacts with many other nodes in the graph. The reason is that we expect to have few bots in our network and therefore if a host shares mutual-contacts with many other hosts, then these mutual-contacts are probably due to a different application other than botnet C&C. Consequently, we compute the dye-attraction coefficient from Node j to Node i as follows:

$$\gamma_{ji} = \frac{E_{ji}}{(D_i)^{\beta}}$$

where D_i is the degree of Node N_i (i.e. number of neighbors or edges that N_i has) and β is the **Node Degree Sensitivity Coefficient**. Basically, nodes with high degrees receive less and less dye as β increases.

The Dye-pumping Algorithm has three inputs, namely the edge capacities (E_{ji}) of the mutual-contacts graph ($\mathbf{E}$ represents the matrix containing all E_{ji} values), the index (s) of the seed node N_s, and the number of iterations ($maxIter$). Given these inputs, the dye-pumping algorithm first computes the dye-attraction coefficients from edge capacities and forms the transition matrix $\mathbf{T}$ such that:

$$\mathbf{T}(i,j) = \gamma_{ji} = \frac{E_{ji}}{(D_i)^{\beta}}$$

where $i = 1, ..., v$ and $j = 1, ..., v$. Also $\mathbf{T}(i,j) = 0$ if $i = j$. Notice that the transition matrix of a mutual-contacts graph with v nodes is a $v \times v$ square matrix.

Following that, the algorithm normalizes $\mathbf{T}$, so that each of its columns sums to 1 (i.e. stochastic matrix). If $\overline{\mathbf{T}}$ indicates the normalized transition matrix, the normalization procedure can be written as $\overline{\mathbf{T}}(i,j) = \frac{\mathbf{T}(i,j)}{\sum_{i=1}^{v} \mathbf{T}(i,j)}$. After normalization, the algorithm iteratively pumps dye to the mutual-contacts graph from the seed node. For this purpose, let the column vector L is the dye level vector, where $L(i)$ indicates the dye level accumulated at node i. The pumping begins with filling the seed node with dye and leaving the others empty such that:

$$L(i) = \begin{cases} 1, & \text{if } s = i \\ 0, & \text{elsewhere} \end{cases}$$

Once the seed node is filled with dye, the algorithm pumps dye from the seed node across the mutual-contacts graph. Since the outgoing edges distribute the dye accumulated within a node proportional to their capacities, the dye levels at next iteration can be computed as:

$$L(i) = \sum_{j=1}^{v} \overline{\mathbf{T}}(j,i) L(j)$$

which can be also written in matrix form as $L = \overline{\mathbf{T}}L$. At each iteration, after updating L, the algorithm pumps more dye to the graph from the seed node by updating $L(s) = L(s) + 1$. Following that the vector L is normalized after each iteration as $L = \frac{L}{\sum_{i=1}^{v} L(i)}$. Finally after $maxIter$ iterations, the dye-pumping algorithm outputs the dye-level vector L. The steps of the dye-pumping algorithm are summarized below:

Algorithm 1 $Dye\_Pumping(\mathbf{E}, s, maxIter)$

1: $\mathbf{T} \leftarrow computeTransitionMatrix(\mathbf{E})$
2: $\overline{\mathbf{T}} \leftarrow normalize(\mathbf{T})$
3: $L \leftarrow [0, 0, ..., 0]^{tr}$ {initialize L as a zero vector}
4: **for** $iter = 1$ to $maxIter$ **do**
5: $\quad L(s) \leftarrow L(s) + 1$ {Pump dye from the seed node}
6: $\quad L \leftarrow \frac{L}{\sum L(i)}$ {Normalize dye level vector}
7: $\quad L \leftarrow \overline{\mathbf{T}}L$ {Distribute dye in network for one iteration}
8: **end for**
9: output L

Once the algorithm outputs the vector L, the dye level of each node ($L(i)$) indicates the confidence level for the corresponding host being a member of the same P2P botnet as the seed node. To have a more conclusive result, we set a threshold thr such that the nodes having a dye level greater than thr are declared as potential members of the same P2P botnet as the seed bot.

Notice that the algorithm involves a constant number of matrix multiplications. Hence, the complexity of a naive implementation of the algorithm is cubic in the number of nodes. However, both dye-level vector (L) and transition matrix ($\mathbf{T}$) are sparse. Therefore one can implement the dye-pumping algorithm asymptotically faster by using fast sparse matrix multiplication techniques.

3. EXPERIMENTS

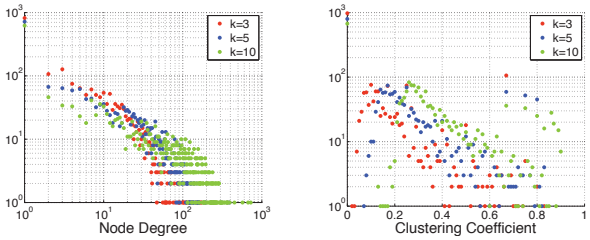
3.1 Detecting Nugache Peers

In order to systematically assess the performance of the proposed scheme against a real-world botnet, one needs to know the IP addresses of the members of a P2P botnet in a given network. Otherwise, nothing can be said about the true positive or false alarm rate without knowing the ground truth. One way to obtain the ground truth is to blend real botnet data into the network traffic and make few hosts look as if they have been infected by the botnet. This strategy essentially aggregates real botnet traffic and real user traffic on some of the hosts and therefore provides a realistic scenario. From the proposed scheme's perspective, to make a host look like a P2P bot, one can first capture the flow records of the network, which contains the host, during a time window. Then one can collect the flow records form a real P2P bot during a similar time window. Following that, one can change the bot's IP address in these botnet flow records to a selected host's IP address and append them to the flow records of the entire network so that, along with its original traffic, the selected host will appear as if it has also communicated with the external IP addresses that the real bot has talked to.

In order to establish the ground truth for our experiments, we utilize the data collected from the Nugache botnet, which has been thoroughly studied in [30][8]. Briefly speaking, Nugache is a P2P botnet that uses random high-numbered ports for its communication over TCP. The data we use in our experiments was compiled by the Nugache crawler presented in [10] and its communication between Nugache peers.

Nugache Botnet Data: Details on the Nugache botnet and Nugache crawler can be found in [30] and [8]. In summary, the C&C protocol of Nugache enables querying a peer for its list of known peers and a list of recently communicated peers. Using this functionality, the crawler starts from a series of seed peers and traverses the botnet by querying peers for their list of known peers. The crawler maintains the list of recently communicated peers for each accessible Nugache peer. Consequently, when it finishes crawling, it produces list of recently communicated peers for several Nugache peers.

In our experiments, we used the data collected by the crawler when Nugache was active. To collect data, the crawler was executed repeatedly for 9 days, where each execution lasted roughly 30 to 45 minutes. We used a 24-hour observation window for our experiments. Hence, we employed several randomly selected 24-hour segments of the crawler data from the 9-day results in our experiments to cover the botnet dynamics during all 9 days. We observed that in any of these 24-hour segments, 904 Nugache peers responded to the crawler on an average. We also observed that 34% of all possible pairs of Nugache peers communi-



(a) Node Degree Dist. (b) Clustering Coeff. Dist.

Figure 3: Node Degree and Clustering Coefficient distributions of the mutual-contacts graph of the background traffic for different privacy threshold (k) values.

cated with at least one mutual-contact on average.

Background Traffic: In order to obtain background traffic that we could blend with Nugache traffic, we captured the flow records observed at the border of Polytechnic Institute of NYU network during a typical weekday (i.e. the observation window is 24 hours). Collected flow records indicate that there were 2128 active IP addresses in our network during the observation window. We then extracted mutual-contacts from the recorded data. To ensure a valid communication (i.e. not a scan flow), we only considered external IPs which exchanged sufficient amount of data (i.e. at least 256 bytes) in both directions with at least one internal IP. Finally we built the corresponding mutual-contacts graph to serve as a basis for our experiments.

We immediately observed in the mutual-contacts graph that DNS servers within the network shared a significantly large number of mutual-contacts with each other. As a matter of fact, DNS servers constituted the highest-magnitude entries of the first eigenvector of the matrix ($\mathbf{E}$) whose entries are the corresponding edge capacities (E_{ij}). This is not surprising since DNS servers in a network communicates with many other DNS servers around the world. Obviously this relationship among DNS servers dominates the mutual-contacts graph and taints the results of the Dye-Pumping algorithm. Hence, we removed all the edges of the 11 DNS servers in the network from the mutual-contacts graph.

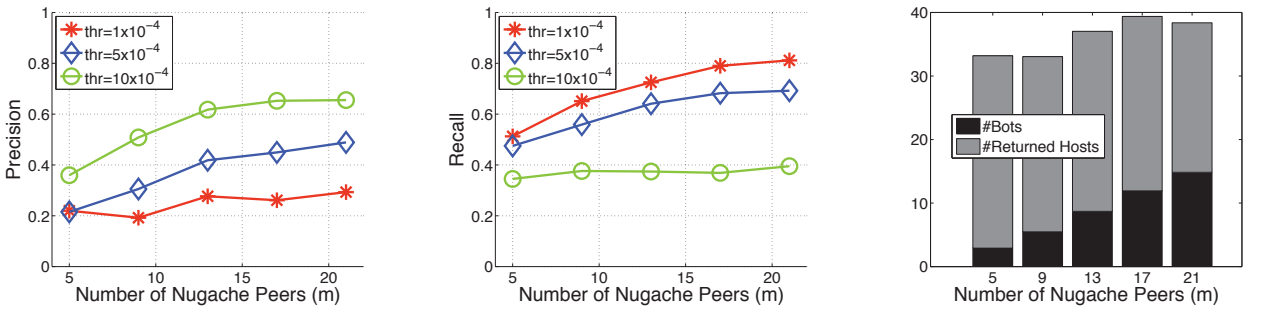
The mutual-contacts graph extracted from the background traffic suggests that majority of the hosts share none or very few mutual-contacts with other nodes. This can be observed in Figure 3(a), where we plot the distribution of node degrees (i.e. no of edge of a node). Figure 3(a) also shows, as expected, that nodes usually have a higher degree in the mutual contact graph when a higher privacy threshold (k) value is used to construct the graph.

We then looked at the clustering coefficient, which is defined as the ratio of the number of the actual edges of a node to the number of all possible edges among it's neighbors. We plot the clustering coefficient distribution of the nodes in Figure 3(b). We observe that the mutual contact-graph is a lot more clustered than a comparable random graph (i.e same number of nodes and edges). For instance the clustering coefficient distribution of a random graph comparable to the mutual-contacts graph with $k = 5$ has a mean of 0.006 and standard deviation of 0.009. This suggests that there are communities of hosts in the observed network where community members usually communicate with the same external IPs that are exclusive to the corresponding community. One can speculate that these communities may represent peers of different P2P networks (legitimate or bot) or a group of users visiting similar websites etc.

Experiments with Nugache: In order to assess the performance of the proposed scheme in detecting Nugache bots, we randomly picked m Nugache peers from a randomly selected 24-hour segment of the crawler data. Then, we computed the mutual-contacts graph corresponding to these m Nugache peers based on the recently-communicated peers field of the crawler data. We then randomly picked m internal hosts from the mutual-contacts graph corresponding to the background traffic. Finally, we superposed the mutual-contacts graph of the Nugache peers onto in the mutual-contacts graph of the background traffic where m Nugache peers coincide with m selected internal hosts. This procedure essentially blends Nugache traffic into the background traffic so that each of these m selected internal hosts looked as if they communicated with the peers that the corresponding m Nugache peers communicated with. Consequently, each of these m selected hosts becomes a real Nugache peer and constitutes the ground truth as far as the proposed scheme is concerned.

Once we obtained the superposed mutual-contacts graph, we randomly selected one of the m hosts as the seed bot and ran the Dye-Pumping algorithm to detect the other $m-1$ hosts whose flow records were modified according to the Nugache crawler data. We set the number of iterations to $maxIter = 5$ for Dye-Pumping algorithm since it is almost impossible to find P2P botnet peers more than 3 hops away from the seed node due to the Erdős-Rényi model as will be explained in Section 4. In the end, we returned the list of hosts which accumulate more dye than the threshold as P2P bots. To obtain statistically reliable results, we repeated the experiment 100 times, each time with different selection of m hosts and m Nugache peers. We also picked a different 24-hour segment of crawler data at every 20^{th} repetition.

Results (Precision & Recall): To gauge the algorithm's performance, we computed the average precision and recall. In our context, precision can be defined as the ratio of the number of Nugache peers in the returned list of hosts to the length of the returned list. On the other hand, recall can be defined as the ratio of the number of Nugache peers in the returned list to the number of all Nugache peers in the network except the seed bot ($m - 1$). Figure 4 presents the average precision and recall values for different number of Nugache peers (m) and different threshold values (thr). We set the privacy threshold $k = 5$ and node degree sensitivity coefficient $\beta = 2$. It is observed that several dormant Nugache peers can be identified by the proposed technique when the threshold is set to an appropriate value. For instance, in Figure 4(c) we observe that, if there are 17 Nugache peers in the network, the proposed scheme on average returns 35 hosts, 11 of which are Nugache peers. As a result, upon obtaining the list of potential P2P bots, a network administrator can perform a more detailed investigation (perhaps physically) on the hosts in the list and potentially uncover several dormant P2P bots. Meanwhile, the returned list also contains some hosts which are not Nugache peers since such hosts happen to be connected to one or more Nugache bots on the mutual-contacts graph due to mutual-contacts created by other applications. Interestingly, it is observed in Figure 4 that both precision and recall values increase as the number of bots (m) increases. This is due to a property



(a) Precision (b) Recall (c) Returned host counts,thr=$5x10^{-4}$

Figure 4: Experiment results with Nugache. The parameters are $k = 5$ and $\beta = 2$

of Erdős-Rényi random graphs that−*as will be explained in the next section*− the probability of having a short path between two nodes increases with the number of nodes. It is also observed that, increasing the threshold increases precision but decreases recall, as is naturally expected for any detection system.

Effects of Privacy Threshold (k): When we repeated the experiments for different k values, we did not observe a major change in the precision performance. On the other hand, we observed, as shown Figure 5(a), that recall performance improves as we decrease k as long as the number of P2P bots in the network is low. The recall performance improves because more background traffic is filtered out for lower k, thereby removing a significant portion of the benign edges. However, if there are many P2P bots in the network and if k is small (i.e. $k = 3$), more than k of them are likely to communicate with several common external peers and therefore some of the botnet communications are likely to be filtered out as well. The effect of this phenomenon can be observed in Figure 5(a), where recall performance diminishes for large number of Nugache peers. Hence, based on Figure 5(a) we can say that $k = 5$ was an appropriate setting for our network.

Effects Node Degree Sensitivity Coefficient (β): As explained in Section 2.3, larger β values result in less dye-flow towards the nodes which have high degrees on a mutual-contacts graph. We would like to restrict the dye-flow to high-degree nodes, because edges between bots and high-degree nodes are probably not due to botnet communications but rather due to some other application which causes many of the edges that high-degree nodes have. Larger β values cause the dye to concentrate around the seed-bot and therefore improve the precision performance as observed in Figure 5(b). On the other hand, since the algorithm cannot reach far in the mutual-contacts graph for larger β values, the recall performance drops as β gets larger as observed in Figure 5(c). According to our experiments, $\beta = 2$ turned out to be an appropriate setting for our network.

In summary, different values of the parameters k and β yield a tradeoff between precision and recall. When deploying the proposed scheme, a network administrator should first decide on the minimum tolerated precision level and then set the parameters accordingly. For this purpose, artificial P2P botnet traffic generated by the Random Peer Selection model described in Section 4.1 could be used as a ground truth to determine which parameter values would result in which precision levels for a given network.

4. MATHEMATICAL ANALYSIS

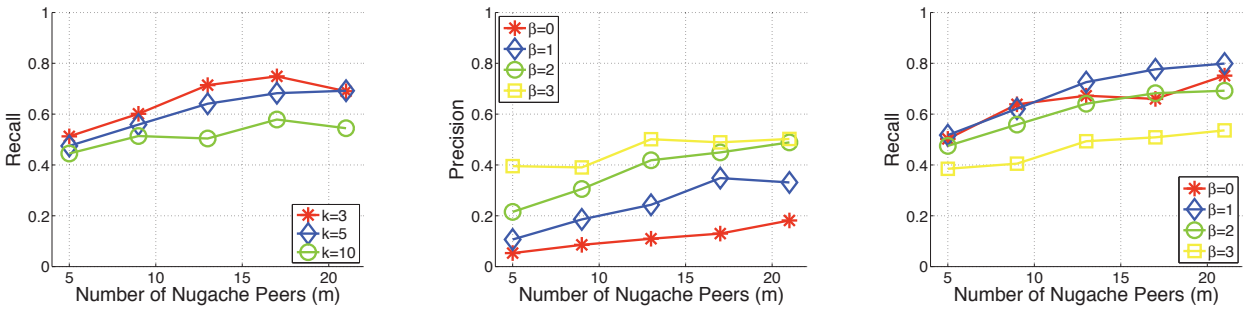
The essence of the proposed algorithm is that the members of a P2P botnet tend to have mutual-contacts and therefore are closely connected on a corresponding private mutual-contacts graph. In fact, the dye-pumping algorithm performs better if P2P bots in a network are connected to the seed node through shorter and higher-capacity paths, which yield higher volume of dye flow from the seed node to the other bots. Although our experimental results in the previous section tend to validate our intuition, some significant questions remain to be addressed to mathematically validate the approach and show its applicability to the general problem that goes beyond specific instances of P2P botnets. Question such as how likely is it that two peer bots will have a mutual contact? How does this probability vary with the size of the botnet and the number of peers contacted by each bot. Next, how likely is it that the mutual contact graph will have a connected component that spans peer bots? How does one characterize the properties of the mutual contacts graph? In this section we address these questions and present a mathematical analysis that supports our approach and validates the experimental results reported in the previous section.

4.1 Random Peer Selection Model

The first question we posed was the likelihood of peer bots having a mutual contact. But before we answer that, we would like to first justify the framework in which we examine this question. Recall that our framework assumes that bots independently and randomly select the peers with which they communicate. How does this assumption bias our analysis? In this subsection we address this question and argue that this represents the worst case situation for our analysis.

In a P2P network some peers might be more available than others and therefore they have a higher probability of being selected by other peers [14][18] [21] [1]. Obviously, having such preferred peers in a P2P botnet increases the chance finding mutual-contacts between P2P bots in a network. However, the worst case, as long as unstructured P2P botnets are considered, from our work's point of view is when there is no preferred peer in the botnet and all peers have equal probability of being contacted by any other peer, thereby minimizing the probability of private mutual-contacts between peers.

To investigate the probability of mutual-contacts in the worst case, we consider a generic botnet model, where each



(a) Recall for different k (b) Prec. for different β (c) Recall for different β

Figure 5: Effects of different parameters. The non-varying parameters are set to $k = 5$, $\beta = 2$ and $thr = 5 \times 10^{-4}$

bot picks peers independently and randomly. The model has two configurable parameters such that; "B" is the number of all peer in the botnet and "C" is the number of peers that each peer communicate with during a specific observation window. Based on these parameters, each bot (b_i) in the model communicates with a uniform random subset ($\mathcal{S}_\mathbf{i}$) of all $B-1$ available bots (excluding itself) in the model, where the cardinality of each subset is C.

Bot-Edge Probability: Having justified our framework, we now address the question about the probability of two peer bots having a mutual contact. In the random peer selection model, the probability of having an edge between two arbitrary bots b_i and b_j (i.e. bot-edge probability, p_e) is actually the probability of the intersection of the corresponding subsets being non-empty; such that $p_e = Pr(\mathcal{S}_\mathbf{i} \cap \mathcal{S}_\mathbf{j} \neq \emptyset)$. Since the number of elements in the intersection of two uniform random subsets can be computed using hypergeometric distribution, we can write the bot-edge probability as:

$$p_e = 1 - \frac{\binom{C}{0}\binom{B-1-C}{C}}{\binom{B-1}{C}} \tag{1}$$

Bot-edge probabilities are plotted in Figure 6(a). It is observed that, similar to the Birthday Paradox, as the number of contacted peers increases, the bot-edge probability increases very rapidly. Consequently, even for a fairly large botnet with $50k$ peers, the bot-edge probability is almost 0.5 when peers contact only 200 other peers during the observation window.

Bot-Edge Capacity: Although, high bot-edge probabilities works in favor of the dye-pumping algorithm, the capacities of those edges are also important. It is obvious that, the higher the bot-edge capacities the better the dye-pumping algorithm performs. In the random peer selection model, the probability of a peer contacted by two given peers is $\left(\frac{C}{B}\right)^2$. Therefore, since there are B peers in total, we can write the expected capacity of bot edges ($E[Cp]$) as:

$$E[Cp] = \left(\frac{C}{B}\right)^2 B = \frac{C^2}{B} \tag{2}$$

which is also the expected value of the corresponding hypergeometric distribution. Figure 6(b) plots the expected bot-edge probabilities. It is observed that, regardless of the botnet size, expected bot-edge capacity rapidly exceeds 1 and continues to increase as the number of contacted peers increases. Figure 6 suggests that the members of a P2P botnet will most probably be well connected with each other on a private mutual-contacts graph through high capacity edges, thereby allowing the dye-pumping algorithm to identify them.

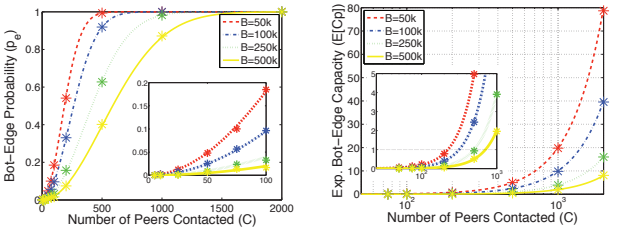
4.2 Friends Stay Closely Connected (Erdős-Rényi Subgraphs)

Having established that it is quite likely that two peer bots will have a mutual contact we now turn our attention on the expected structure of the mutual contacts graph. After all, the Dye-Pumping algorithm can only identify the P2P bots which are connected to the seed-bot via short paths on the mutual-contacts graph. Bots which are isolated from the seed-bot cannot be accessed by the algorithm. In this subsection, given a bot-edge probability, we investigate how the P2P bots are expected to be oriented on a private mutual-contacts graph and what portion of the P2P nodes can be accessed by the dye-pumping algorithm.

To understand the structure of the subgraph formed by members of a P2P botnet on a mutual-contacts graph, suppose that there are m bots in the network, and therefore the corresponding m nodes on the graph. Let the set $X = \{X_1, X_2, ..., X_m\}$ denote these nodes and p_e denote the probability of having an edge between any given X_i and X_j, for $i \neq j$ where $1 \leq i \leq m$ and $1 \leq j \leq m$. Since p_e is the same for any pair of X_i and X_j, the subgraph formed by the nodes$X_1, X_2, ..., X_m$ on a private mutual-contacts graph is an *Erdős-Rényi random graph* [11][12], where each possible edge in the graph appears with equal probability.

One interesting property shown by Erdős and Rényi is that, Erdős -Rényi graphs have a sharp threshold of edge-probability for graph connectivity [12]. More specifically, if the edge-probability is greater than the threshold then almost all of the graphs produced by the model will be connected. Erdős and Rényi have shown the sharp connectivity threshold is $\frac{\ln \theta}{\theta}$, where θ is the number of nodes in the graph. Therefore, if the bot-edge probability of a P2P botnet is $p_e = \frac{\ln m}{m}$, then the dye-pumping algorithm potentially identifies all other P2P bots from a given seed bot with high probability as long as there are more than m bots in the network. In other words, it gets easier for the proposed method to reveal P2P bots as the botmaster infects more hosts in the network. However, even if the bot-edge probability is below the threshold, the dye-pumping algorithm can still identify some of the P2P bots, which happen to be connected to the seed node on the private mutual-contacts graph.

In conclusion, according to the random peer selection model, members of a P2P botnet are expected to be closely connected to each other on a private mutual contacts graph despite large botnet sizes.



(a) Bot-Edge probability (b) Expect.value of bot-edge capacity

Figure 6: Properties of random peer selection model for different botnet sizes (B) and different number of contacted peers (C) are plotted in Figure 6(a). Solid lines indicate the theoretical computation and the stars point the empirical estimation. Inner figures magnifies the region where $0 < C < 100$

5. LIMITATIONS AND POTENTIAL IMPROVEMENTS

The proposed method is able to identify P2P bots in a network as long as they are clustered through short and high capacity paths on a private mutual-contacts graph. Therefore, botmasters need to disturb this clustering structure in order to evade the proposed method. In this section, we review these possible evasion strategies, and their implications on the creation and maintenance of P2P botnets.

Eliminating Private Mutual-Contacts: One way to eliminate private mutual contacts is by increasing the popularity of private mutual-contacts that P2P bots in a network communicate with. If their popularity gets higher than the privacy threshold (k), they will be omitted by the proposed scheme and will not result in edges in private mutual-contacts graphs. However, in order to achieve this, a botmaster has to control more than k hosts in that particular network, so that they can collectively boost a contact's popularity beyond the privacy threshold. To defend against this strategy, the privacy threshold (k) needs to be set as large as possible. Although, as discussed in Section 3, high k values impairs the recall performance of the proposed scheme, for smaller networks it is often possible to find an appropriate k value since a botmaster is unlikely to have too many bots in a small network. On the other hand, for large networks which potentially contain many P2P bots, the proposed technique can be applied on smaller subnets separately and independently to increase the likelihood that the number of P2P bots in each subnet remain below the privacy threshold.

Decreasing The Probability of Mutual-Contacts: Decreasing the probability of observing mutual-contacts between P2P bots is equivalent to decreasing the bot-edge probability (p_e). As discussed in Section 4, a botmaster has to either(or both) increase the botnet size (B) or decrease the number of peers that each bot communicates with (C) in order to lower p_e. It is clear that increasing B and decreasing C will inversely affect a P2P botnet's robustness and efficiency. Although it may be possible for a botmaster to pull p_e down to a lower value, we observed in a controlled environment that peers of today's botnets such as Storm and Waledac have very high bot-edge probabilities. To collect data for Storm and Waledac, we infected two Pentium IV, 512MB RAM Windows XP hosts, which were completely isolated from the rest of the network by a firewall. The firewall was also set to block all SMTP traffic to prevent any spam traffic. We observe that both Storm and Waledac communicate with fairly high number of unique peers during 24 hours, and therefore create many mutual-contacts as presented in Table 1. On the contrary, Nugache peers are less active and create far less mutual-contacts as observed in Table 1. Nevertheless, in Section 3, the proposed scheme is shown to successfully detect several Nugache peers, which are introduced to the network using the crawler data, despite their low communication activities. To collect data for Nugache, the bots were installed on a Pentium IV, 1GB RAM, running VMware Server with a Windows XP guest, as well as on bare metal machines on comparable hardware running Windows XP. The traces were captured within the protected network using a customized honeywall [32] and also using full-packet capture on an extrusion prevention system running OpenBSD with strict packet filter rules, as described in [10] The captured packets were converted to flow records using the SiLK tools [4] for establishing mutual contact sets and validating the algorithm.

Table 1: Summary of observed P2P botnet behavior. Δ : Average number of unique IP addresses that a bot communicates with each day. ◯ : the number of mutual-contacts (the bot-edge capacities) between the two bots during 24 hours.

	Day 1		Day 2		Day 3	
	Δ	◯	Δ	◯	Δ	◯
Storm	5180	2861	4681	2886	4022	2323
Waledac	1145	341	775	300	1012	358
Nugache	45	0	53	1	49	0

Using a Structured P2P Topology: A botmaster can adopt a structured P2P topology to decrease the probability of mutual contacts by making peers in a same network to communicate with different sets of peers from each other. To achieve this, peers in a same network have to coordinate with each other so that they won't communicate with the peers in each other's peer list. In some sense, peers in a same network have to form their own tiny botnet among themselves and appear as a single node to the remaining of the P2P botnet. These intra-network communications among the peers in a same network, however, would potentially yield new means of detecting P2P bots in a network. Nevertheless, even if a botmaster manages to deploy a mutual-contact-free P2P architecture, two or more networks can choose to share their flow records to exploit the mutual-contacts among P2P bots in different networks, which are unavoidable since the botmaster cannot know which networks would collaborate in the first place. For such mitigation strategies, cooperating networks can use privacy-preserving set operations such as [7] to share data between networks without revealing any sensitive information.

Poisoning Clusters: The purpose of cluster poisoning for P2P networks is to destroy clustering structure of a graph by creating bogus edges [5]. Cluster poisoning appears to be very hard to achieve in our context. In order to perform poisoning, a botmaster has to create an edge between a P2P bot and a benign node on a mutual-contacts graph. For this purpose, she needs to make both the bot and the benign host communicate with a mutual external IP. To do so, the botmaster has to listen to the traffic of the benign host and make the P2P bot contact with an external host which the benign host has communicated with. But this is not a trivial task for a botmaster, unless she also possesses a router or a

proxy in the same network.

6. RELATED WORK

Early botnets employed centralized *command and control* (C&C) servers to distribute commands and updates to individual bots, usually through IRC or HTTP protocols [9]. Although a centralized structure is simple and easy to manage, it suffers from a single point of failure and is susceptible to traditional defenses such as domain revocation, DNS redirection, blacklisting etc. Therefore, botmasters have begun to adopt a P2P architecture for C&C channels. In [20], authors argue that it is harder to detect P2P bots especially with a limited view of their traffic from a single Autonomous System. In P2P botnets each bot acts both as a server and a client allowing botmasters to publish commands and updates from any point in the botnet[14][18]. In [6], authors investigate the effects of different botnet structures.

There have been numerous techniques proposed to detect botnets. In [25] and [24], the authors employ machine learning techniques where they train classifiers with different features to detect botnet C&C flows. In [31], Strayer *et. al.* proposed a technique to detect botnet activity by exploiting temporal correlations between C&C activities of the bots belonging to the same botnet. Binkey and Singh proposed a technique to detect IRC botnets in [2] using botnet-related anomalies in TCP and IRC statistics. Another IRC botnet detection scheme was proposed by Goebel and Holz in [13], where the authors exploited the structure and evolution of IRC nicknames used by IRC bots. In [23], Karasaridis *et. al.* combined traffic aggregates with IDS alarms to identify centralized botnets within a Tier-1 ISP. In [16], Gu *et. al.* proposed BotHunter, which searches for a specific pattern of events in IDS logs to detect successful infections caused by centralized botnets.

All the above schemes were designed to detect either specific botnets that they were trained for, or centralized botnets. In general, detecting P2P bots in a network is harder since there is no trivial correlation that allows us to link together the P2P bots in a network, especially when bot peers communicate with each other using encryption and through random ports [14, 18, 10].

As a completely different problem from ours, crawler based methods were proposed to enumerate P2P bots globally in [22] and [18]. Since crawlers cannot enumerate P2P bots behind NAT and/or firewall in [21] Kang *et. al.* proposed a sybil attack based passive monitoring scheme to enumerate P2P bots even behind NAT or firewall. However, P2P bot enumeration methods are not intended to identify local P2P bots in a network. Also, they require implantation of bot peers which requires reverse engineering of a bot binary and its C&C protocol.

Coming back to our problem, there have been few techniques proposed which are able to detect local P2P bots assuming that P2P bots exhibit similar malicious activities and similar connection patterns. In [17], Gu *et. al.* proposed BotSniffer to detect bots based on spatial-temporal correlation between bot responses to commands. Following that, in [15], Gu *et. al.* proposed BotMiner which clusters the hosts in a network by their malicious activity and communication patterns. Their results showed that members of a botnet usually fall within the same cluster. Similarly, in [33], Yen and Reiter proposed a scheme called TAMD, where traffic containing similar external IPs, similar payloads and similar internal platform types are aggregated to detect botnets in a network. Although clustering based botnet detection schemes are successful in detecting many current P2P bots, botmasters can evade them by assigning different tasks to the bots in the same network or by randomizing their communication patterns as acknowledged in [15]. In [29], authors systematically investigate such evasion techniques. Also, clustering based schemes fall short in detecting idle P2P bots which haven't exhibited any overt behavior yet.

7. CONCLUSION AND FUTURE WORK

In this paper, we presented a simple and efficient method to identify local members of a P2P botnet in a network, starting from a known member of the same botnet in the same network. The basic idea of the proposed method is that, the members of a botnet are more likely to have mutual-contacts with each other than with benign hosts. We evaluate the proposed method using real P2P botnet (Nugache) data captured by a crawler. We also provide a mathematical analysis of the C&C structure of P2P botnets to characterize the performance of the proposed method. Both our analysis and experiments show that the proposed scheme is able to identify several dormant P2P bots in a network.

There are some limitations of the proposed scheme as discussed in Section 5. Perhaps the most important one is that, a botmaster can evade detection if she employs a structured P2P topology which ensures that her bots avoid mutual-contacts while communicating with each other. However, developing such a mechanism is not trivial for today's botnets and currently available P2P topologies. Nevertheless, even if a botmaster achieves such a topology, two or more networks can mitigate this by sharing their network traffic, possibly in a privacy-preserving manner, to exploit the mutual-contacts which will possibly occur between peers in different networks. We leave the exploration of the benefits of data-sharing as future work. In addition, we plan to study on a new P2P botnet architectures, which potentially evade the proposed scheme at least in some scenarios. This will allow us to further improve the proposed scheme to withstand potential evasion strategies, which might be employed by next generation botnets.

8. ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their comments and David Dittrich for his valuable contributions.

9. REFERENCES

[1] R. Bhagwan, S. Savage, and G. M. Voelker. Understanding availability. In *The 2nd International Workshop on Peer-to-peer systems*, 2003.

[2] J. R. Binkley and S. Singh. An algorithm for anomaly-based botnet detection. In *SRUTI'06: Proceedings of the 2nd conference on Steps to Reducing Unwanted Traffic on the Internet*, 2006.

[3] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM Conference on Computer and Communication Security*, Chicago, IL, November 2009.

[4] CERT Coordination Center. SiLK: System for internet-level knowledge. Available at `http://tools.netsa.cert.org/silk/`.

[5] D. R. Choffnes, J. Duch, D. Malmgren, R. Guierma, F. E. Bustamante, and L. Amaral. Swarmscreen:

Privacy through plausible deniability in P2P systems. Technical report, Northwestern EECS Technical Report, March 2009.

[6] D. Dagon, G. Gu, C. Lee, and W. Lee. A taxonomy of botnet structures. In *Proceedings of the 23 Annual Computer Security Applications Conference (ACSAC'07)*, December 2007.

[7] L. K. Dawn and D. Song. Privacy-preserving set operations. In *in Advances in Cryptology - CRYPTO 2005, LNCS*, pages 241–257, 2005.

[8] D. Dittrich and S. Dietrich. Discovery techniques for P2P botnets. In *Stevens Institute of Technology CS Technical Report 2008-4*, September 2008.

[9] D. Dittrich and S. Dietrich. New directions in peer-to-peer malware. In *Sarnoff Symposium, 2008 IEEE*, April 2008.

[10] D. Dittrich and S. Dietrich. P2P as botnet command and control: A deeper insight. In *MALWARE 2008. 3rd International Conference on Malicious and Unwanted Software*, 2008.

[11] P. Erdos and A. Renyi. On random graphs I. *Publ. Math. Debrecen 6*, pages 290–297, 1959.

[12] P. Erdos and A. Renyi. The evolution of random graphs. *Magyar Tud. Akad. Mat. Kutato Int. Kozl 5*, pages 17–61, 1960.

[13] J. Goebel and T. Holz. Rishi: identify bot contaminated hosts by IRC nickname evaluation. In *HotBots'07: Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, 2007.

[14] J. B. Grizzard, V. Sharma, C. Nunnery, B. B. Kang, and D. Dagon. Peer-to-peer botnets: overview and case study. In *HotBots'07: Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, 2007.

[15] G. Gu, R. Perdisci, J. Zhang, and W. Lee. BotMiner: Clustering analysis of network traffic for protocol- and structure-independent botnet detection. In *Proceedings of the 17th USENIX Security Symposium (Security'08)*, 2008.

[16] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. BotHunter: Detecting malware infection through ids-driven dialog correlation. In *Proceedings of the 16th USENIX Security Symposium (Security'07)*, August 2007.

[17] G. Gu, J. Zhang, and W. Lee. BotSniffer: Detecting botnet command and control channels in network traffic. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, February 2008.

[18] T. Holz, M. Steiner, F. Dahl, E. Biersack, and F. Freiling. Measurements and mitigation of peer-to-peer-based botnets: a case study on Storm Worm. In *LEET'08: Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, 2008.

[19] M. Iliofotou, P. Pappu, M. Faloutsos, M. Mitzenmacher, S. Singh, and G. Varghese. Network monitoring using traffic dispersion graphs (TDGs). In *IMC '07: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 315–320, 2007.

[20] M. Jelasity and V. Bilicki. Towards automated detection of peer-to-peer botnets: On the limits of local approaches. In *Proceedings of the 2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats LEET'09*, April 2009.

[21] B. B. Kang, E. Chan-Tin, C. P. Lee, J. Tyra, H. J. Kang, C. N. Z. Wadler, G. Sinclair, N. Hopper, D. Dagon, and Y. Kim. Towards complete node enumeration in a peer-to-peer botnet. In *Proceedings of ACM Symposium on Information, Computer and Communications Security (ASIACCS 2009)*, March 2009.

[22] C. Kanich, K. Levchenko, B. Enright, G. M. Voelker, and S. Savage. The Heisenbot uncertainty problem: challenges in separating bots from chaff. In *LEET'08: Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, pages 1–9, 2008.

[23] A. Karasaridis, B. Rexroad, and D. Hoeflin. Wide-scale botnet detection and characterization. In *HotBots'07: Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, 2007.

[24] S. Kondo and N. Sato. Botnet traffic detection techniques by C&C session classification using svm. *Advances in Information and Computer Security*, pages 91–104, 2007.

[25] C. Livadas, R. Walsh, D. Lapsley, and W. Strayer. Using machine learning technliques to identify botnet traffic. *Local Computer Networks, Annual IEEE Conference on*, 0:967–974, 2006.

[26] S. Nagaraja, P. Mittal, C.-Y. Hong, M. Caesar, and N. Borisov. BotGrep: Finding P2P bots with structured graph analysis. In *USENIX Security Conference*, August 2010.

[27] P. Porras, H. Saidi, and V. Yegneswaran. Conficker C P2P Protocol and Implementation, September 2009. `http://mtc.sri.com/Conficker/P2P/`.

[28] G. Sinclair, C. Nunnery, and B.-H. Kang. The waledac protocol: The how and why. In *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*, pages 69 –77, October 2009.

[29] E. Stinson and J. C. Mitchell. Towards systematic evaluation of the evadability of bot/botnet detection methods. In *WOOT'08: Proceedings of the 2nd conference on USENIX Workshop on offensive technologies*, 2008.

[30] S. Stover, D. Dittrich, J. Hernandez, and S. Dietrich. Analysis of the storm and nugache trojans: P2P is here. In *;login: The USENIX Magazine*, volume 32-6, December 2007.

[31] W. Strayer, R. Walsh, C. Livadas, and D. Lapsley. Detecting botnets with tight command and control. *Local Computer Networks, Annual IEEE Conference on*, 0:195–202, 2006.

[32] The Honeynet Project. Honeywall, 2009. `https://projects.honeynet.org/honeywall/`.

[33] T.-F. Yen and M. K. Reiter. Traffic aggregation for malware detection. In *DIMVA '08: Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 207–227, 2008.

The case for in-the-lab botnet experimentation: creating and taking down a 3000-node botnet

Joan Calvet
École Polytechnique de Montréal, Canada

Carlton R. Davis
École Polytechnique de Montréal, Canada

José M. Fernandez
École Polytechnique de Montréal, Canada

Jean-Yves Marion
LORIA, Nancy, France

Pier-Luc St-Onge
Ecole Polytech. de Montréal

Wadie Guizani
LORIA, Nancy, France

Pierre-Marc Bureau
ESET
Montréal, Canada

Anil Somayaji
Carleton University
Ottawa, Canada

ABSTRACT

Botnets constitute a serious security problem. A lot of effort has been invested towards understanding them better, while developing and learning how to deploy effective counter-measures against them. Their study via various analysis, modelling and experimental methods are integral parts of the development cycle of any such botnet mitigation schemes. It also constitutes a vital part of the process of understanding present threats and predicting future ones. Currently, the most popular of these techniques are "in-the-wild" botnet studies, where researchers interact directly with real-world botnets. This approach is less than ideal, for many reasons that we discuss in this paper, including scientific validity, ethical and legal issues. Consequently, we present an alternative approach employing "in the lab" experiments involving at-scale emulated botnets. We discuss the advantages of such an approach over reverse engineering, analytical modelling, simulation and in-the-wild studies. Moreover, we discuss the requirements that facilities supporting them must have. We then describe an experiment in which we emulated a 3000-node, fully-featured version of the Waledac botnet, complete with an emulated command and control (C&C) infrastructure. By observing the load characteristics and yield (rate of spamming) of such a botnet, we can draw interesting conclusions about its real-world operations and design decisions made by its creators. Furthermore, we conducted experiments with sybil attacks launched against it and verified their viability. However, we were able to determine that mounting such attacks is not so simple: high resource consumption can cause havoc and partially neutralise them. Finally, we were able to repeat the attacks with varying parameters, in an attempt to optimise them. The merits of this experimental approach is underlined since by the fact that it would have been difficult to obtain these results by other methods.

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

1. INTRODUCTION

Botnets constitute one of the most worrying computer security threats. Practically all Internet users have experienced the ill effects of botnets, whether by receiving large volumes of spams daily, having their confidential information stolen, lost access to critical Internet services, etc. Botnets are complex and large distributed systems consisting of several thousands, and in some cases, millions of computers. In order to develop a good understanding of such a distributed system and gain insights on its vulnerabilities and weaknesses, it is necessary to study the system as a whole. To that purpose, efforts need to be made to understand how the various parts of the system interact, and in particular how the size and scale of such systems affect their performance.

While analysis by reverse engineering of botnet binaries can initially help us better understand them, it does not always provide the "big picture" in terms of botnet operations. This is because its other parts might not be visible or accessible. Beyond that, we can partially increase our understanding by observing and analysing in-the-wild botnets as a whole, giving indirect visibility of some of these inaccesible components. Studies such as [6, 15, 16, 18, 22] conducting experiments with in-the-wild botnets, have contributed to furthering understanding of botnets. Nonetheless, this method can be problematic, owing to the following: (i) In order to experiment with in-the-wild botnets, researchers need to create entities which join the botnets and perform the tasks the researchers stipulate. If a significant number of entities is added to a botnet, it is possible that the botnets operators will detect the presence of these entities, and possibly implement counter-measures to protect their botnets, and in so doing, potentially shift the botnet arms race further in their favour. On the other hand, if the number of such entities introduced constitutes only a small portion of the overall botnet, we might not be able to accurately observe or predict at-scale effects. (ii) There are legal and ethical issues involved in performing in-the-wild botnet research; for example, in some jurisdictions (particularly in Europe), it is considered unethical and even illegal to create entities that join a botnet, despite the fact that their purpose might be to disrupt the botnet. There are also potential risks involved in doing in-the-wild botnet research: some researchers who investigated botnets have reported that their domains have received distributed DoS attacks from the botnets [12, 21]. (iii) It is difficult to get statistically significant results for experiments involv-

ing in-the-wild botnets. Values that are ascertained for variables via a single experiment run—which often require several weeks or months to complete—may be outliers rather than being representative values. In principle, the only way to guarantee that the results are statistically significant is to repeat the experiment multiple times until the standard deviation of the values are within acceptable limits. As highlighted above, it may be undesirable or even counter-productive to perform an experiment on in-the-wild botnets multiple times. Nonetheless, statistical significance is very important because the changing conditions of the environment (churn of infected population, actions by humans, etc.) could give the appearance that, for example, a mitigation strategy is effective, even though the experimenter just happened to be "lucky" at the time of the experiment. (iv) Since in-the-wild botnets experiments are not controlled and (normally) cannot be repeated, they do not allow us to explore the full design space and potential choice of parameters, for example, those related to mitigation strategies being developed. Thus, the solution tested and validated in a single in-the-wild experiment could be far from optimal. For example, a failed experiment because of an unlucky bad choice of parameters could lead us to believe that a promising approach will not work, and cause us to prematurely abandon it . Simulation studies and analytical modelling have also been employed for botnet investigations. Analytical models are often complex, and all but the more simplistic models are hard to understand and resolve. While mathematical models like Markov chains [3] and immunological equations have been used for other kinds of malware, e.g. worms, botnet-specific analytical modelling has been less common, either addressing propagation properties [8], performance of the C&C infrastructure using graph theory [11, 25], or other techniques [19]. Simulation results, on the other hand, are more accessible and can be obtained by using ready-made network simulators such as Opnet/Omnet, ns2, etc. and adapting them to specific protocols, or by home-coding special purpose discrete-event simulators tailored to model a particular botnet (e.g. the Kademlia/Storm simulator in [9,10], and generic botnet models in [7, 23]). However, while it is easier to more precisely measure these performance criteria in simulations, this approach has the disavantadge that all aspects of the botnet must either be modelled and implemented, or simply modelled away and ignored. This is particularly problematic for two reasons. First, except for finer-grained, network-based simulators (and even then), it is hard to model and appropriately reproduce the network transmission characteristics of the Internet. Second, it is also quite hard to model and simulate the behaviour of the universe of infectable machines and users, which is particularly important in understanding the "churn" within the botnet due to infection/disinfection, power-on-power-off cycles, etc.

For these reasons, at-scale emulation studies, where conditions as close as possible to the real-world are reproduced in a controlled environment, are perhaps the best alternative to in-the-wild studies. Emulation studies allow controlled repetition of the experiments to see whether variations in environmental parameters, whether these are controlled (by experiment design) or uncontrolled (but measurable) variables, significantly affect performance results. Moreover, they are paramount in threat prediction research, in that they allow us to safely explore the botnet design space in scenarios where the botnet operating parameters have been optimised, something that would be unthinkable with in-the-wild experimentation. On the other hand, in comparison with simulation studies, where artificial models are used in lieu of real botnet entities, in emulation experiments, botnet entities that are either identical or slightly adapted versions of their real-world counterparts, are executed in controlled environments. While this at-scale approach requires large amount of system resources and experimental preparation efforts, it is worth pursuing due to its many advantages. First, as mentioned above, this approach allows researchers to have greater control over the experimental environment; consequently, more thorough investigation encompassing greater variation of experiment parameters can be undertaken. Second, botnet emulation experiments can provide information about botnets that would be very difficult or virtually impossible to ascertain via in-the-wild studies, via simulation experiments, or via reverse engineering analyses. Third, evaluating botnet mitigation schemes using emulated botnets rather than in-the-wild studies, allows researchers the privilege of hiding their ammunition from botnets operators, until the mitigation schemes are fully developed and optimised, at which point, the schemes can be made available to appropriate authorities or those who feel "the calling" and have the resources, and clout or mandate to overtly go on the offence against botnets. Fourth, in addition to facilitating more thorough evaluation of botnet mitigation schemes (as highlighted above), emulation studies can be conducted in significantly less time than in-the-wild studies. Therefore, with this approach, security researchers and practitioners can be more effective and proactive in the fight against botnets. Finally, at-scale botnet emulation provides an avenue for investigating botnets that does not present the same level of legal and ethical issues involved in actively investigating botnets in-the-wild.

For all of these reasons, we jointly endeavoured to develop a different approach for conducting such at-scale, in-the-lab botnet emulation experiments, as an alternative to these other methods of botnet analysis. In this paper, we introduce the philosophy, methodology and tools of this approach, and present a case study involving a particular botnet. The experiment described herein involved recreating in the lab an isolated version of the Waledac botnet [4, 20] consisting of approximately 3,000 nodes, and further, testing and validating a mitigation scheme against it (sybil attack), that we had theorised was possible in such previous work. The specific contributions of the paper are the following: (i) we introduce and show the feasibility of recreating and studying isolated at-scale botnet in a secure environment, (ii) we provide the first significant evidence that the Waledac botnet is vulnerable to sybil attack by demonstrating it in the lab, (iii) we illustrate how such emulated botnets can be used to validate, refine and optimise botnet takedown mechanisms, and (iv) we illustrate how at-scale experimentation of this type can be used to obtain otherwise unaccessible information by revealing previously unknown details about the non-visible components and design decisions taken by Waledac creators and operators.

The rest of the paper is structured as follows. Section 2 describes some previous work in the construction of experimental platforms supporting botnet research. We then discuss the criteria that this type of platforms should meet in order to support at-scale botnet emulation experiments in a safe and scientifically sound manner. We also describe the testbed and generic tools that we have used to conduct our 3000-node botnet emulation experiment with the Waledac botnet. Waledac itself and the experiment are described in Section 3, where we also discuss the results obtained. This includes both results about the viability of the sybil attack we described in previous work [4] and, more interestingly, some valuable insights regarding Waledac design and operations, that could not have been obtained by other methods. We discuss the relevance of these results with respect to validating this kind of experimental approach in Section 4 and conclude in Section 5 by summarizing our contributions, presenting some limitations of our work and highlighting avenues for future research.

2. BOTNET EMULATION EXPERIMENTS

2.1 Related Work

The idea of using laboratory experimentation facilities for botnet research is not new. PlanetLab [17], Emulab [24], and DETER [2] are popular network testbeds that are based on computers hosted at multiple facilities. DETER in particular is specially geared towards security research. These experimental platforms, though they have proven to be very valuable facilities for researchers, are not that suitable for high risk security experiments, such as botnets emulation, owing to the risk of malware "breaking" through logical barriers and escaping into the wild.

With regards to work related to high risk security experiments, a botnet evaluation environment is described in [1] that is a "plug-in" for Emulab-enabled network testbeds. This work is an initial step in building a scalable laboratory testbed for experiments with botnets, but one of the approaches the authors have used to contain the network traffic within the testbed is to give the nodes unroutable (private) IP addresses, which severely limits the type of experiments that can be run on the testbed. Moreover, they only managed hundreds of malicious bots, thus not allowing at-scale emulation of large modern botnets. Jackson *et al.* [13] use DETER to deploy their System for Live Investigation of Next Generation bots (SLINGbot) which, according to the authors, "enables researchers to construct benign bots for the purposes of generating and characterizing botnet command and control traffic". We took a quite different approach mainly because we wanted to run high risk experiments, e.g. involving real malware binaries, and thus we decided to totally isolate our environment from the Internet.

Finally, John *et al.* [14] created a platform named Botlab which monitors the behaviour of spam-oriented bots. Some of the goals of this work is similar to ours, they are both geared to studying botnets. However, there are significant differences. In their work, real-world in-the-wild botnets are monitored, while in ours a complete botnet is reproduced in an isolated and secure environment.

2.2 Design Criteria

The two computer security research labs involved in this work have both adopted stringent security rules and scientific criteria. This is a requirement in order to be able to conduct safe and relevant experimental security research in general, and botnet emulation experiments in particular. A full description of these facilities and the associated criteria is given in [5]. We reproduce here the criteria that we consider are specifically applicable to botnet emulation experiments.

Highly secured. Malware can be potentially highly contagious and is (by definition) developed for malicious intents. Consequently, experiments involving malware should therefore take adequate precautionary measures to ensure that it is not accidentally released into the wild during the course to the experiment. Perhaps the only way of adequately mitigating risk associated to this threat is for the experiment environment to be completely isolated from the Internet and other networks. Thus we build our emulation platform based on an isolated cluster within highly secured facilities. The physical security of the labs includes strong physical barriers (floor-to-ceiling walls, reinforced doors, etc.), surveillance systems (cameras, motion detectors), a separate access control system using multi-factor authentication. In terms of logical security, the cluster is completely isolated from other computer networks (air gapped).

Scale. We desire to have an emulation platform capable of supporting-at scale experiments; i.e. involving several thousands of bots. The choice followed was to heavily rely on virtualisation. This allowed us to have upwards of 30 virtual bots per physical machine.

Realism. An important requirement of our botnet emulation platform is that it be capable of reproducing botnets that in principle are identical (or close to identical) in functionality to those found in the wild. To achieve this, it is necessary that very few changes (or ideally none at all) be made to the bot binaries that are used to reproduce the botnets. Changes should be restricted to those that are necessary (if any are required) to overcome anti-virtualisation and anti-debugging capabilities in the bot binaries. This constraint necessitates that nodes in the emulation platform be configured with IP addresses that are hard coded in the bots binaries, and that the necessary DNS databases be setup to resolve these addresses.

Flexibility. We desire to have an emulation platform that is capable of reproducing any botnet after the necessary reverse engineering and investigative work has been done to elucidate the structure of the botnet command and control. Therefore, flexibility is an important requirement. The emulation platform should be easily configurable to adapt various overlay network topologies with for example, variable proportions of bots with private (unroutable) IP addresses versus bots with public IP addresses: proportions that mirror those observed in the in-the-wild botnet.

Sterilisability. To guarantee the integrity of the experiments, virtual machines (VM) need to be "sterilised" in order to remove any artifacts associated with the malware infection. In certain cases, this requires removal and re-installation of the VMs. Efficient mechanisms are therefore needed to accomplish these tasks.

2.3 Hardware and Tools

In order to meet these criteria, we used an isolated cluster as our emulation platform. The cluster consists of 98 blades, each having a quad-core processor, 8 Gb of RAM, dual 136 Gb SCSI disks and a network card with 4 separate gigabit Ethernet ports. The blades are contained in two 42U racks. The blades are interconnected with two separate sets of switches (each in their own 42U rack) such that two physically separated networks are created: a) a *control network* used for transmitting commands and data necessary for controlling the experiments, and b) an *experiment network* used for transmitting the experiment traffic, including in this case botnet activity (spam, C&C traffic, etc.). Having two physically separated networks helps to guarantee the integrity of the experiment, in that, commands and data necessary for controlling the experiment can be sent via a separate network. This ensures that the control traffic does not interfere with the transmission of the experiment traffic, thus preserving the validity of timing measurements made on it.

We present a brief overview of the virtualisation and configuration management tools we employed.

Virtualisation: To maximise the versatility and capability of the emulation platform, we sought a feature rich virtualisation technology that is able to emulate both Windows and Linux. Consequently, we choose the VMWare ESX product as the hypervisor for the blades, which allows good efficiency and ease of configuration.

Configuration and management: We used the Extreme Cloud Administration Toolkit (xCAT), an open-source tool, for configuring and managing the emulation platform. xCAT is particularly attractive for this purpose since it contains VMWare functionalities; for example, xCAT can create defined number of VMs with a single command, such as, `mkvm vm[001-098]`, thus creating 98 VMs which are assigned names vm01, vm02, ..., vm098. From a management point of view, xCAT operates as follows. First it requires tables containing host configuration information, including details such as machine template (i.e. location and name of the ghost image), hostname, IP address, etc. These tables can be filled manually using a text editor or they can be generated using perl or any other scripting language. When the tables are filled, xCAT can be issued commands causing the tables to be committed to the xCAT database. It incorporates powerful image deployment, con-

figuration and control commands, that take the information from the database, and use remote boot technology such as PXE or the ESX API, to order hosts to do the required tasks. Thus, the experiment design, deployment and management process for emulated experiments is as follows. First, xCAT tables must be filled to facilitate the deployment and configuration of appropriate host images containing ESX. Following this, the researchers produce an abstract, high-level description of the desired environments, and build necessary VM templates or ghost images (e.g. a VM template for each type of bot, gateways, SMTP servers, etc.). Next, the researchers decides on a network topology, addressing plan and host naming convention. xCAT tables then need to be filled to facilitate the deployment and configuration of these entities (ESX hosts, VMs, and their configurations). Depending on the size of the experiment, xCAT tables can be filled manually or automatically using scripts, regular expressions or a combination of both.

2.4 Experiment Methodology

Generally speaking in order to prepare, design and conduct an at-scale botnet emulation experiment (some or all of) the following steps are followed:

1. Capture of botnet client code, through various methods (honeypot, collaborators, etc.).
2. Gather information on the botnet in order to understand as much as is possible about the botnet architecture and modes of operation. Examples of information that are required are (i) communications protocols and message formats; (ii) authentication process for gaining access to the botnet; (iii) categories of bots and the hierarchical relationship between them; and (iv) C& C architecture. This information can typically be obtained by reverse engineering bots and analysing their communication traffic.
3. Passively monitoring the botnet by observing infected machines and/or joining the botnet with special purpose passive botnet-like programmes (crawlers), in order to continue to gain information on its structure, in particular the C&C infrastructure, including formats of commands, location and characteristics of C&C servers, etc.
4. Construction of a surrogate C&C infrastructure complete with servers and any intermediary proxies.
5. Construction of realistic *operating environment* for the botnet in the lab, including infectable/infected machines (ideally showing human driven-like behaviour), ancillary network services (DNS, SMTP, DHCP, etc.), a realistic emulated network architecture, and, of course, counter-measures and mitigation schemes against it.
6. Determination of metrics to be measured, based on research questions that experiment must answer.
7. Implementation of methods for measuring these metrics and extracting the results in usable form for further analysis.

The main challenges in following this methodology involve:
Maintaining isolation. This means both a) maintaining spatial and logical isolation between the experimental and control components (achieved in our setup through physical separation), and b) maintaining isolation between the whole facility and the outside world (security criterion). In addition, it also means time and logical separation between successive experiment runs (sterilisability criterion).
Observing without interference. This is paramount in order to maintain the scientific soundness of the results, and also to enforce isolation.
Simulating network characteristics and user behaviour. This is a very hard problem that is not just relevant to botnet research. It is essentially a modelling problem combined with significant engineering issues.

3. THE WALEDAC EXPERIMENT

To exemplify this methodology, we now describe how we applied it in constituting an isolated Waledac botnet and launching our sybil attack against it.

3.1 Overview of Waledac

Waledac is a prominent botnet which first appeared in November 2008, shortly after the Storm botnet became inactive. Waledac employs a "home grown" peer-to-peer (P2P) network infrastructure for its C&C. Other researchers and members of our group [4, 20] had previously reversed engineered the Waledac binary and obtained details about its mode of operation. Here, we will limit our description of Waledac to the aspects relevant to the goals of this research, i.e. disruption of its C&C through sybil attacks.

Waledac botnet uses a four layered C&C architecture. The first layer contains bots that are referred to as *spammers*. These are machines with private IP addresses residing behind Network Address Translator (NAT) devices. Spammers are essentially the "worker" bots and constitute approximately 80% of the botnet. Their principal role is to send spam, harvest email addresses from files stored on the infected machines, and harvest confidential information (e.g. usernames and passwords) from the network traffic that traverses the infected machines.

Waledac binaries are hardcoded with a list consisting of 100 to 500 contact information of *repeaters*. This list—which is referred to as a *RList*—is stored in XML format in a registry key. An *RList* has a global Unix timestamp and between 100 to 500 records that contain the following fields: a 16-byte ID, an IP address, a port number, and a Unix timestamp. The list is sorted in descending order of timestamp (oldest at the top of the list). The *RLists* play a key role in facilitating Waledac operations and maintaining the P2P infrastructure, in the following ways: (i) The *Rlist* allows each node to "know" a small subset of the botnet nodes. A spammer, for example, contacts the repeaters in its *RList*, at frequent intervals, and request "jobs" (i.e. tasks to perform). The repeater will in turn, forward the job request to a *protector*, which will subsequently forward the request to the C&C server, and relay the response (the work order) to the repeater, which will issue the work order to the spammer. (ii) The *RList* provides a means of propagating identification information of repeaters which recently join the botnet. This process is facilitated as follows. All bots regularly send update messages to their known repeaters. For a bot S that wishes to send an update message we have two possibilities: a) if S is a spammer, it extracts 100 records from its *RList* and sends this extract to a randomly selected repeater; and b) if S is a repeater, it selects 99 records from its *RList*, adds its own record (containing its identification information and the current timestamp) to the top of the list, and sends the list to the selected repeater. When the recipient bot R receives the list, it reciprocates the process, by sending a list containing 100 records of repeaters it knows of, back to bot S. The recipient of an update list uses the list to update its *RList*, as indicated below in Section 3.3.

In addition to an *RList*, each repeater also has a cryptographically-signed protector list, containing identification information on the protectors. The repeaters regularly exchange signed protector lists.

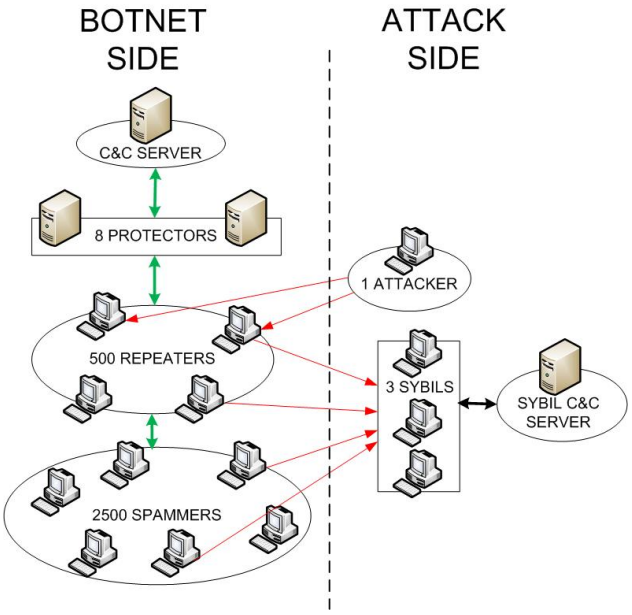


Figure 1: Experimental setup

The private key for signing such lists is known only to the C&C server, and the public key certificate for verifying them is embedded in the bot code. Note that it would be nonsensical to sign *RLists* since any bot (even if infiltrated) must be able to provide them. More interestingly, commands from the server are not signed either, something that could provide some level of protection against a sybil attack. However, the traffic between the server and the bots is encrypted with AES-128, using a key that is chosen by the server (and was probably meant to be a session key).

Waledac also provide a failback mechanism that allows bots to maintain connection to the botnet even if the repeaters listed in its *RList* are not reachable. The failback mechanism works as follows: if a bot makes 10 consecutive unsuccessful attempts to contact a repeater, the bot connects to a HTTP server (the URL for the servers are hardcoded in the Waledac binary) and download an updated *RList*. These lists are updated every 10 minutes on the server, so that they contain the most recently "heard of" repeaters.

3.2 Waledac emulation

The overall setup and architecture for our emulation experiment involving a contained Waledac botnet is depicted in Figure 1. The process we employed to constitute it is as follows:
(*i*) *Create VM templates*. First, we installed the binaries on Windows XP VMs and created xCat VM templates associated with them. We created separate templates for spammer and repeaters.
(*ii*) *Add the IP addresses of 500 repeaters to the RLists*. We deleted the entries in the original *RLists*, and added the identification information of the 500 repeaters we used for the experiment.
(*iii*) *Add script to issue commands to the VMs*. We created a Python script and added it to the VM template. This script allows us to issue commands to the VM, for example, to start and stop execution of the Waledac binaries, to clean the VMs, to restore the *RList* to its initial state, and delete the *RList* dumps (see Section 3.4).
(*iv*) *Deploy the VM templates*. Next, we utilised xCAT to install the VM templates on the blades (approximately 30 VMs per blade).
(*v*) *Setup C&C server*. Through our in-the-wild investigation of Waledac, we were able to determine the type and the format of the messages the C&C server sends to the bots in response to those it receives from them. The server code is a Python script that is capable of responding to all such requests in a similar manner as the original C&C server. For example, we observed that spam orders issued to bots contain between 500 and 1,000 email addresses. We mimic this functionality by creating five different spam order messages, each containing between 500 and 1,000 addresses and programmed the C&C to send them to bots requesting spam jobs. We also implemented the failback scheme as follows: every 10 minutes the script creates an *RList* containing the identification information of the most active repeaters; this list is placed on a HTTP host. All HTTP requests from the Waledac binaries to the hardcoded failback domains are directed to this host. As is the case for Waledac C&C server, our C&C server utilises 1024-bit RSA and 128-bit AES keys to provide confidentially services for the messages the C&C server and the bots exchange. The server runs on a VM that is the only one to run on that blade.
(*vi*) *Constitute the botnet*. Finally, we issue commands to the VMs to start running the Waledac binaries. We can similarly stop and re-start the experiment at will. The botnet we constituted for our experiment consisted of 500 repeaters, 2,300 spammer, 8 protectors and the C&C server (for a total of 2,809 nodes in the botnet). This proportion is close to that which is observed in the wild for Waledac. It should be noted that the protectors we created are actually components of the C&C server, and not separate machines: we assigned 8 network interfaces—each with a different IP address—to the C&C server for this purpose. These addresses are set to be identical to those of the real Waledac protectors. This was necessary because the protectors identification information hardcoded in the bots binaries is signed and we have no knowledge of the corresponding private key.
(*vii*) *Setup environment*. In addition to the blades in the cluster, we used standalone Linux machines to setup the ancillary infrastructure needed for the botnet to run. These standalone machines provide services, such as DNS, SMTP and DHCP, that would normally be present in the Internet. They constitute a simple reproduction of part of the "environment" within which the real botnet would operate. These machines were of course connected to the experiment network of the cluster.

3.3 Mitigation scheme and implementation

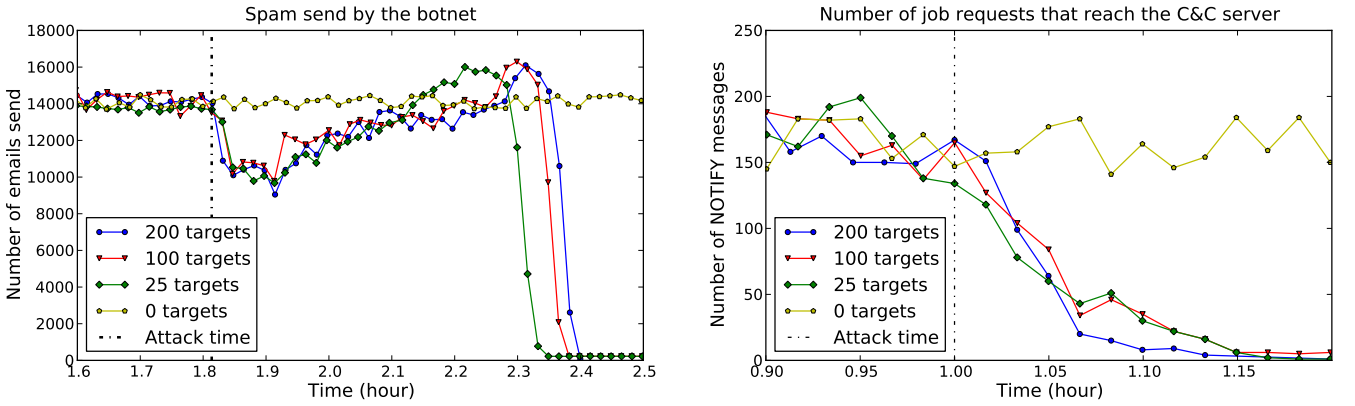
By reverse engineering the Waledac code and analysing its network traffic, we had previously conjectured [4] that Waledac was vulnerable to sybil attacks, due to characteristics of the home-made P2P protocol it uses for C&C. In addition, because the IP address of a bot needs not be unique (bots are primarily identified by their 16-byte ID), it is possible to generate large number of sybils—with unique IDs but with the same IP address—whilst using few machines, thus making this attack relatively easy to mount.

We indicated in Section 3.1 that bots use the update messages they receive, containing a 100-entry extract of the sender's *RList*, to update their *RList*. For each entry i in the update list, the recipient computes a new timestamp:

$$NewTS_i = CurrentTS - |UpdateTS - TS_i|$$

where *CurrentTS* is the current timestamp, *UpdateTS* is the list's global timestamp and TS_i is the timestamp of the entry. The recipient then replaces the timestamp and inserts the entry in its *RList* at the correct location: recall that an *RList* is sorted in descending order of timestamp. All entries beyond position 500 are deleted.

By analysing the binaries, we also discovered that Waledac bots do not check the update lists they receive to determine if they contain more than 100 records. It is therefore possible to craft special



(a) Spam output before and after sybil attack

(b) Job requests arriving at the C&C server before and after attack

Figure 2: Botnet activities before and after sybil attack being launched.

update messages that will cause all the entries in the *RLists* of the recipients to be entirely replaced by the sybil records. This can easily be accomplished by placing 500 sybil records in the update list and set the timestamp of all the sybil entries, to a value that is identical to the list's global timestamp. This guarantees that the *NewTS* for all the sybil entries will be equal to the current timestamp; consequently these 500 sybil entries will be placed at the top of the recipient's *RList* and all the others will be deleted.

The extent to which a bot can be controlled and isolated when it receives such a message, depends on the type of bot.

If the bot is a repeater. After receiving the message from the sybil, a race condition situation arises. Since the repeater's identity information is likely to be in other bots' *RLists*, they are likely to send the repeater update messages whose entries could replace some of the sybil entries in its *RList*. To maximise the chances that the repeater remains completely isolated, the sybils need to continue sending update messages to the repeater at short time intervals.

If the bot is a spammer. In this case the result of the sybil attack is more effective, since the spammer cannot be contacted directly. When it receives an update from a sybil, and the entries in its *RList* are consequently completely replaced by sybil entries, the spammer will become completely isolated from other bots. It should be noted though, that in order to infiltrate a spammer's *RList*, the sybils first need to infiltrate the *RLists* of repeaters whose identity information are in the *RList* of the spammer. Also, in order to mantain isolation, the sybils need to remain active until all the Waledac domains that are encoded in the affected bots are disactivated, otherwise the bots will resort to the failback mechanism we described in Section 3.1, and download a "clean" *RList* from a Waledac HTTP server.

For the sybil attack implementation, we used three separate entities: (a) fake repeaters (sybils), (b) attackers, and (c) a sybil C&C server. The role of the sybils is to passively respond to update messages—sent to sybils—with responses containing the specially crafted update message we described above, whereas the attackers' role is to target specified numbers of repeaters and send them the specially crafted update messages. The records in the update messages all contain sybil ID information. The attackers send these messages to the targeted repeaters once every minute. This rate was utilised because we observed that the Waledac bots in-the-wild send between 2 to 5 update messages during a two-minute time period. The role of the sybil C&C server is to prevent the "turned" bots from resorting to the failback mechanism. We indicated in Section 3.1 that if a bot makes 10 consecutive unsuccessful attempts to contact repeaters, the bot will connect to a Waledac Web server and download an updated *RList*. In order to prevent this from happening, the sybils are programmed to relay all messages from the turned bots to the sybil C&C server. The sybil C&C server will in turn send harmless spam orders to the turned bots. We use the following feature of Waledac to issue these harmless orders: spammers are supplied with a special SMTP server IP address that they are required to use to test if they are capable of sending spams. Before sending spam, spammer try to connect to this special SMTP server and send spam only if they succeed. We therefore send the bots the address for an "SMTP server" that is unreachable. In so doing, we can ensure that the bots the sybils control, do not send spam.

We employed three VM to host the sybils, one VM to launch the attackers and another VM to run the sybil C&C server.

3.4 Experiment results

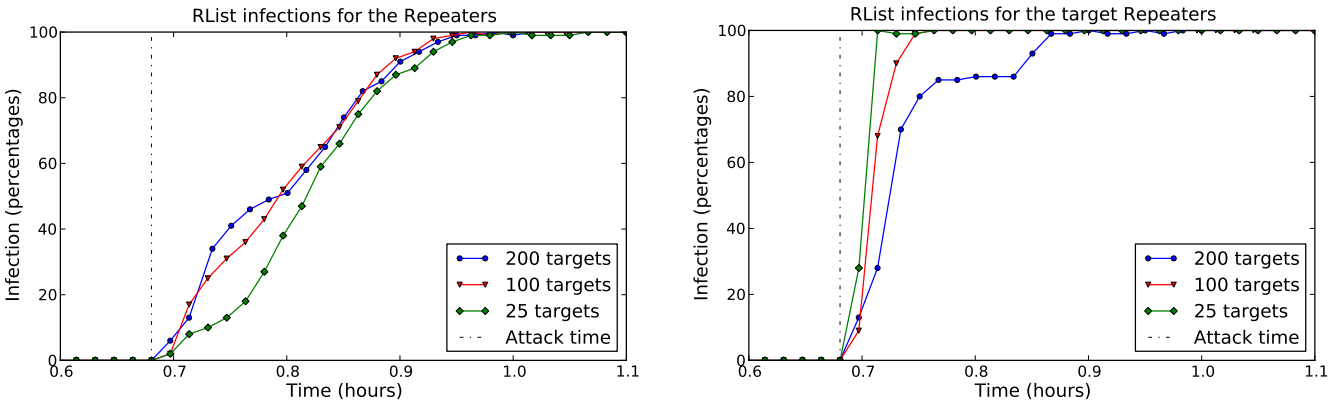
We utilised the following metrics for assessing the effectiveness of the sybil attack mitigation scheme.

Spam output. We measure the spam output of the botnet, over a fixed time period, before and after we launch the attack. To facilitate spam output measurement, we programmed our botnet C&C server to send spam orders with email addresses belonging to the same domain. This allows us to more easily count the number of spam sent by counting DNS requests to that domain.

Connectivity of the botnet. In order to determine the extent to which the sybil attack affects the connectivity of the botnet, we measure the number of NOTIFY messages the C&C server receives over a fixed time period, before and after we launch the attack. NOTIFY messages are the second message that a bot sends when it dialogues with the C&C server. Counting only NOTIFY messages allows us to filter out noise due to failed connection requests.

Percentage of sybils in *RList*. The goal of the attack is to replace the entries in *RLists* with sybil records, which will utlimately isolate the bots. This parameter is thus an intermediary indicator of effectiveness. We measure it by way of a Python running on the bots, that dumps *RList* to a file each time it is modified, and send these files to an FTP server via the control network. We then analyse these files to determine the percentages of sybils in the *RList*.

In addition to measuring the above metrics, we also wish to determine the degree of success of the attack when subsets of known repeaters are targeted, vice targeting all known repeaters. We consequently performed 3 sets of experiments, targeting 200, 100 and



(a) Percentage *RList* infection for all repeaters

(b) Percentage *RList* infection for targeted repeaters

Figure 3: Percentage *RList* infections by sybils.

25 repeaters, respectively.

The first set of experiments was used to benchmark the botnet. We performed 3 experiment runs; for each run, we allow the botnet activities to reach a steady state, then we measure spam output and botnet connectivity. The average values—for the 3 experiment runs—for these measures were 13,200 emails per minutes and 120 NOTIFY messages per minute, respectively.

The next set of experiments assessed the efficacy of the sybil attack by determining the steady state values of the above metrics, before and after the attack begins. We performed 3 experiment runs for each set of experiments and compute the average values for the the runs. As indicated in Table 1, the standard deviation values obtained are relatively small, and we therefore believe that these results are statistically significant.

Figure 2(a) shows the spam output of the botnet before and after the sybil attack begins. The dotted vertical line indicates the time that the attack begins. The graph shows that the attack is a success and that the spam output drastically falls after less than an hour. We also observe that after the sybil attack begins, there is an initial decrease in the spam output, with a subsequent gradual return to its original level, and even rising significantly above it, before the final fall. Furthermore, we can see that the time taken to reach the final fall is longer with 200 and 100 targets than it is with 25, which is also not so intuitive.

We think the main cause of these two surprising and interesting effects is the load on the C&C servers, both the sybil and the malicious one. The C&C servers are overloaded and cannot keep up with the computing time required for cryptographic operations. As mentioned in [4], the Waledac botnet uses RSA with 1024 bits keypairs and AES-128. Through our observation of the Waledac botnet in-the-wild, we discovered that the C&C server used the same AES session key for all bots, for approximately 10 months. We initially thought that this was a design error made by the botnet creators, but when implementing the Waledac C&C server we discovered that it was not impossible to generate a session key for each bot, because it overloads the server with cryptographic computation. Waledac bots are too verbose and if a good availability of the C&C server is desired, there is no choice but to keep the same session key for all active connections (at least for several minutes) and give bots the same set of encrypted orders. Hence it is likely that this was no mistake, but rather a conscious design choice by Waledac creators.

However, as the sybil attack progresses, the sybil C&C server has fewer cryptographic operations to perform because we use exactly the same strategy as the Waledac C&C server in-the-wild: we use the same session key and pre-encrypt the work orders in batch mode before they need to be sent. Thus as the attack progresses the sybil C&C server availability does not decrease too significantly since it does not have to encrypt any orders, hence allowing it to control an increasing number of bots, and adequately handle their requests.

It is important to note that there is a delay between the moment we completely infect a spammer's *RList* and the moment it stops sending spam: a bot will not contact the C&C server until it has finished its current task.

As the attack progresses and we gain a hold within the botnet, we decrease the load on the real C&C server, which becomes more available for the non-poisoned bots. Thus, these bots receive orders every time they ask (which is not a normal situation, even in-the-wild) and continue to spam in a more efficient way than in a normal state. During that short interim time period, the botnet is more efficient under attack than in its normal state. It should be noted that if we had given more resources to the C&C server to start with, this effect would probably be less important, but as we attributed 4 processors and 8 Gb RAM for its VM, we think it is realistic to assume that this effect would also be observed in-the-wild.

Figure 2(b) shows the number of NOTIFY messages the C&C server receives before and after the sybil attack begins. The number of NOTIFY messages the C&C server receives is essentially a measure of the connectivity of the botnet. The figure indicates that, as expected, there was a gradual decrease in the number of messages that arrive at the C&C server, after the sybil attack commences. After the attack begins, the more efficient attack is the one with 25 targets. This is also a consequence of the load on the sybil C&C server. Because it is overwhelmed with the more aggressive attacks, it refuses connections. After a transition period, the 100 and 200 targets attacks become eventually more effective as the sybil C&C server has fewer cryptographic computation to perform.

Figure 3(a) shows the percentage of sybils entries in all repeaters *RList* (targets and non-targets). After an initial transitory phase where the more aggressive attacks (more targets) seem more efficient, we reach a stage of equivalent linear growth in the number of sybil-controlled machines in the *RLists*, This is due to the fact that propagation of sybil records in the *RList* of non-sybil, non-targeted bots is dependent on the rate of *RList* updates between non-targeted real bots, which is the same for all attacks.

Figure 3(b) shows the percentage of poisoning on the targeted repeaters only. We can observe that it is quicker to fully control

25 direct targets than 100 or 200, because of the race condition faced by these direct targets: the more bots we target, the higher the chance that we lose some races and have sybil records replaced by real ones.

4. DISCUSSION

As previously discussed, at-scale botnet emulation in the lab displays several advantages with respect to other analysis methods. The Waledac experiments that we have conducted exemplifies the viability of this approach and provides clear indications of some of these advantages.

First, we were able to assess the efficacy of the mitigation scheme directly by measuring the following three parameters: (i) the number of NOTIFY messages arriving at the C&C server within a given time period, (ii) the number of spam sent within a given time period; and (iii) the penetration ratio of sybil identification within the bots peer lists (*RLists*). These parameters provide the most effective means of measuring the connectivity and productivity of the botnet. Whereas we were easily able to measure these parameters via such botnet emulation experiments, the value of these parameters are virtually impossible to ascertain—particularly items (i) and (iii)—via in-the-wild botnet studies.

Second, we were able to address and answer questions about attack optimisation. In particular, in our attack the role of fake repeaters (the sybils) is to target a specified subset of repeaters by sending them specially crafted update messages. An important question that needed to be answered regarding the implementation of the mitigation scheme is, what is the ideal number of repeaters to target? It is very difficult to design in-the-wild botnet experiments to find answer to this question. Moreover, even if it were possible to do so, these experiments would likely take several weeks or even months to complete, whereas botnet emulation experiments supply the answer to this question within a few hours.

Moreover, some of the experiment results seem counter-intuitive. They indicate, for example, that targeting higher number of repeaters does not necessarily cause the efficacy of the mitigation scheme to increase. By performing the experiment multiple times and observing the same trend, it became clear that the larger the number of repeaters that are targeted, the more update requests will be sent to the fake repeaters (sybils) that respond to update messages sent to sybils; and if the number of update messages sent to the sybils increase beyond a given threshold, many of these messages will be dropped and consequently will not be serviced. This leads to higher number of repeaters that are under the control of the sybils resorting to the failback mechanism (as outlined in Section 3.1) and download "clean" *RList*, and in so doing, breaking free of the control of the sybils. Becoming aware of this fact is important for a couple of reasons. Firstly, it provides pointers as to how the mitigation scheme can be made more stealthy, since in targeting smaller number of repeaters it is likely that the probability of the botnet operators detecting the presence of the counter-botnet agents in the botnet will decrease. Secondly, this awareness provides indicators as to how the counter-botnet agents can allocate their resources to maximise the efficacy of the counter-botnet operations. In essence, we were able to discover this phenomenon (repeaters re-joining the real botnet because of sybil overload) by running an at-scale botnet emulation experiment where we could observe and note the behaviour of bot clients. Again, it would have been very difficult to notice this by passive in-the-wild botnet observation, unless researchers have machines that join the botnet and play an active role in them (i.e. send spam and support criminal activities), something that many would consider dangerous and questionable. Thus, this constitutes a third example of why at-scale botnet emulation are a necessary tool in botnet research.

Since fake repeaters responding to update messages sent to targeted repeaters were easily overloaded, we can deduce that this task requires more resources than the attackers, whose role it is to send specially crafted update messages to the targeted repeaters in order to place fake repeaters in their *RList*. We did not directly address the question of what would have been the optimal value for the ratio of fake repeaters to attackers, i.e. how to best allocate sybil machines to these roles. However, by running the experiments multiple times, it became clear that the resources we allocated for responding to update messages sent to the fake repeaters were inadequate, since large number of messages that were addressed to the fake repeaters were dropped, simply because the service queue was being filled. Whereas this observation could also have been made via in-the-wild botnet experiments, it is much easier to verify via in-lab experiments such as this.

Finally, the last example has nothing to do not with the attack, but rather with the botnet and the botmasters themselves. By not only directly observing the bot clients but also the reconstructed botnet C&C server, we were able to "wear the shoes" of the botmaster and were thus able to identify some of the performance and design challenges that botnet creators and botmasters must face. In particular, what would have been a textbook solution to ensure data confidentiality and integrity, i.e. the use of unique symmetric keys for each session between a bot and the C&C server, turned out to be non viable due to the size of the botnet. Without at-scale experimentation in the scale of several thousand bots, we would never have discovered this fact. This illustrates that, surprising to some (including some of us!), one can indeed learn a lot about the bad guys even in the lab. In other words, field work is not by itself the end-all of botnet and cyber criminality research.

5. CONCLUSIONS

In this paper we presented an alternative approach for conducting botnet research: at-scale botnet emulation in laboratory conditions. We have discussed its generic advantages with respect to other approaches like analytical modelling, simulation studies, and in-the-wild botnet experimentation. In a nutshell, it provides a greater verifiable realism than analytical models, simulation methods or small-scale emulations, while providing greater levels of control and safety, and presenting fewer ethical and legal problems than in-the-wild experimentation.

In order to deliver such advantages, however, botnet emulations must be run on platforms or testbeds that meet certain criteria. We have postulated and described such necessary criteria. Namely: i) security, to mitigate risks of accidental or unauthorised release of botnet code or information about them; ii) scalability, in order to be able to emulate botnets of large enough size so that similar phenomena as those in a real botnet can be observed; iii) realism, for the same reason; iv) flexibility, so that experiments can easily be repeated, under varying controlled conditions and for different types of botnets and/or mitigation schemes, and v) sterilisability, so that results from previous experiments do not affect that of future ones.

Using the isolated security testbeds based on virtualisation [5], we were able to mount a set of at-scale emulation experiments of the Waledac botnet involving close to 3,000 bots. The controlled conditions of the lab and the full visibility on the botnet and the ancillary infrastructure (the botnet's "operating environment") allowed us to measure performance metrics for both the botnet and attacks agains it that would have been very hard to measure in in-the-wild botnet, such as i) spam yield (i.e. number of spams per minute sent by the bots), ii) botnet activity (i.e. number of NOTIFY messages per minute), and iii) penetration of sybils into the botnet

(a) Average (30 min period before attack) and Std deviation values for spam output

Experiment	Average	Standard deviation for the spam output each 2 minutes after the start of the attack														
25 targets	13219	403.27	389.29	102.75	149.51	101.22	528.81	182.02	121.48	211.20	230.79	274.75	169.00	445.75	325.85	395.44
100 targets	13433	180.07	546.72	195.69	327.22	22.61	326.74	271.37	250.24	339.77	338.79	511.07	187.33	171.66	315.68	462.08
200 targets	13582	506.29	348.47	144.76	312.90	769.54	56.01	493.12	239.83	450.14	160.21	662.21	378.59	154.85	406.08	562.96

(b) Average (30 min period before attack) and Std deviation values for job request reaching the C&C

Experiment	Average	Standard deviation for the number of job requests that reach the C&C server														
25 targets	185	23.33	19.09	7.78	3.54	1.41	0.71	7.78	5.66	4.95	0.71	0.71	1.41	0.71	0.00	0.00
100 targets	176	19.22	39.59	21.63	5.03	2.52	3.00	6.03	0.00	0.58	0.00	0.58	0.58	0.00	0.00	0.58
200 targets	172	25.70	12.22	5.51	10.02	16.07	4.58	7.00	2.31	4.04	1.73	0.58	0.58	0.00	0.00	0.58

(c) Std deviation values for percentage *RList* infection for all the repeaters

Experiment	Standard deviation for percentages of *RList* infection each 2 minutes during the transition state														
25 targets	0.00	1.00	2.52	0.58	2.31	2.31	1.53	2.65	2.65	0.00	0.58	0.58	0.00	0.00	0.00
100 targets	0.00	0.50	0.82	2.22	1.50	2.38	1.63	2.63	2.00	1.26	1.00	0.50	0.00	0.00	0.00
200 targets	0.00	1.65	2.08	1.73	1.53	5.00	0.58	2.08	2.65	1.73	1.00	0.58	0.00	0.00	0.58

(d) Std deviation values for percentage *RList* infection for the target repeaters

Experiment	Standard deviation for percentages of *RList* infection each 2 minutes during the transition state														
25 targets	0.00	1.97	2.90	0.71	0.71	1.54	0.00	2.83	0.00	2.83	0.00	2.12	2.12	0.00	1.41
100 targets	0.00	1.53	0.58	2.00	0.58	0.58	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
200 targets	0.00	4.78	3.49	2.87	2.36	2.12	0.00	0.00	0.00	0.00	1.24	1.83	0.00	0.00	0.71

Table 1: Standard deviation values for the experiment runs

(i.e. percentage of sybils in bot peer lists). The results obtained by measurement of these quantities can be summarised as follows. With respect to the efficacy and viability of the sybil attack, we can conclude that:

1. The sybil attack as implemented is indeed effective and achieves full disruption within an hour.
2. Workload on the sybil C&C server is an important factor to consider, as it creates a transient "window of detection" that could allow the botmaster to detect the attack before he completely loses control of the botnet.
3. It is not necessary to poison the *RList* of all or even many repeaters for the attack to succeed. In fact, targeting smaller numbers of repeaters (as few as 5% of them) yields essentially the same disruption results as wider attacks (targeting up to 40% of the repeaters), while somewhat mitigating the workload problem of the sybil attack C&C server.

With respect to the actual botnet, we were able to deduce the following facts from our results:

4. Workload on the actual C&C server is also a problem. We suspect that this is the real reason why common session keys started to be used 10 months into the botnet deployment, and not due to a programming or design mistake, as was initially suspected. This is also the probable reason why server commands are not signed.

It is very important to note that it would have been very difficult, and in some cases impossible, to reach these same conclusions by resorting to other methods of botnet analysis. While the efficacy of the sybil attack on the real botnet could have been measured by continuous monitoring the attacked botnet, it is unlikely that the attack designer would have had a chance to run several experiments to find out that limited targeted attacks are a better option. In addition, and as mentioned above, testing counter-measures in-the-wild could bave several negative side-effects (retaliation, premature disclosure of mitigation strategies, premature beginning of an arms race, etc.) that could easily outweigh the benefits of such research. In addition, without running an actual C&C server in the lab for an at-scale reproduction of the botnet, it would not have been possible to confirm that the design choices made by the botnet creators were due to performance issues. This cannot be deduced from real-world botnet observation, unless one has gained access to the actual C&C server, a very unlikely proposition.

Thus, we hope to have made a strong case for the use of at-scale botnet emulation as a fundamental tool in botnet research, complementary at least, and superior in many respects to other botnet and counter-measure study techniques. Nonetheless, there are some important limitations to this approach.

First, they require access to testing facilities that meet the above-mentioned criteria; this is unfortunately not the case today for many good and well-established botnet researchers. National and international collaborative efforts like those in which the authors are involved, or the US DETER project are one way to address this. However, even though it is understandable that actual usage of the facilities might be restricted, more collaboration and sharing of procedures, tools and standards would greatly benefit the community as a whole and encourage researchers and research funders to follow that path.

Second, while we were very careful in the fidelity of the botnet emulation portion of our experimental setup, the emulation of the operating environment of the botnet is somewhat simplistic. Aspects of the environment that can be included in that category are: i) a more realistic model and emulation of the Internet (including Layer 3 and below characteristics such as topology, latency, adressing, etc.) as it interconnects the bots, the C&C server and the ancillary infrastructure, ii) a more realistic model describing the natural oscillations in botnet population —also referred to as *churn* or *birth-death process*— due to user action such as infection/disinfection, powering on and off, diurnal usage patterns, etc. Internet networks and user modelling is another field of research all on its own, and a very hard one at that. Nonetheless, we are currently working on ways to easily and transparently port and implement such given models to our security testbeds, which would

allow us to test the impact of changes in network configuration and user behaviour on botnet and counter-measure efficacy. This, we hope, will lead to very fruitful research, as we, the good guys, do in principle control the network and can positively affect user behaviour through education or regulation.

Finally, none of our experiments emulate the behaviour of an important part of the botnet: the botmaster, who deploys and operates the botnet and that has, in principle, clearly defined objectives for doing so (e.g. profit). While it is not as easy to capture the "botmaster code" into the lab as it was for the bot code itself, it would be relatively easy to adapt our botnet emulation to allow for interactive "gaming" of a botmaster vs. botnet attacker scenario, where both are played by security researchers in real time or off-line by surrogate "game" engines that play out pre-defined strategies. This approach would allow us to quantify the typical payoff matrices that are used in game theory to try to predict the ultimate outcome of such scenarios.

Acknowledgments

This research was partially funded by Canada's Natural Sciences and Engineering Research Council (NSERC) strategic research network on Internetworked System Security Network (ISSNet). We are also very grateful for the valuable input and feedback we received from Patrick McDaniel on previous versions of this manuscript.

6. REFERENCES

[1] P. Barford and M. Blodgett. Toward botnet mesocosms. In *Proc. 1^{st} Work. on Hot Topics in Understanding Botnets (HotBots)*, Apr. 2007.

[2] T. Benzel, R. Braden, D. Kim, C. Neuman, A. Joseph, K. Sklower, R. Ostrenga, and S. Schwab. Experience with DETER: A testbed for security research. In *Proc. IEEE Conf. on Testbeds and Research Infrastructures for the Dev. of Networks and Communities (TridentCom)*, Mar. 2006.

[3] P.-M. Bureau and J. Fernandez. Optimising networks against malware. In *Proc. Int. Swarm Intelligence and Other Forms of Malware Work. (MALWARE)*, Apr. 2007.

[4] J. Calvet, C. Davis, and P.-M. Bureau. Malware authors don't learn, and that's good! In *Proc. Int. Conf. on Malicious and Unwanted Software (MALWARE)*, Oct. 2009.

[5] J. Calvet, C. Davis, J. Fernandez, W. Guizani, M. Kaczmarek, J.-Y. Marion, and P.-L. St-Onge. Isolated virtualised clusters: testbeds for high-security experimentation and training. In *Proc. 3^{rd} USENIX Work. on Cyber Sec. Experimentation and Test (CSET)*, Aug. 2010.

[6] E. Cooke, F. Jahanian, and D. McPherson. The zombie roundup: Understanding, detecting, and disrupting botnets. In *Proc. Work. on Steps to Reducing Unwanted Traffic on the Internet (SRUTI)*, July 2005.

[7] D. Dagon, G.Gu, C.Zou, J. Grizzard, S. Dwivedi, W. Lee, and R. Lipton. A taxonomy of botnets. In *Proc. of CAIDA DNS-OARC Work.*, July 2005.

[8] D. Dagon, C. Zou, and W. Lee. Modeling botnet propagation using time zones. In *Proc. 13^{th} Network and Distributed System Security Symp. (NDSS)*, Feb. 2006.

[9] C. Davis, J. Fernandez, and S. Neville. Optimising sybil attacks against P2P-based botnets. Proc. 4^{th} Int. Conf. on Malicious and Unwanted Software (MALWARE), Oct. 2009.

[10] C. Davis, J. Fernandez, S. Neville, and J. McHugh. Sybil attacks as a mitigation strategy against the storm botnet. In *Proc. 3^{rd} Int. Conf. on Malicious and Unwanted Software (MALWARE)*, Oct. 2008.

[11] C. Davis, S. Neville, J. Fernandez, J.-M. Robert, and J. McHugh. Structured peer-to-peer overlay networks: Ideal botnets command and control infrastructures? In *Proc. 13^{th} European Symp. on Research in Computer Security (ESORICS)*, Oct. 2008.

[12] S. Gaudin. Storm botnet puts up defenses and starts attacking back. `http://informationweek.com`, Aug. 2007.

[13] A. Jackson, D. Lapsley, C. Jones, M. Zatko, C. Golubitsky, and W. Strayer. Slingbot: A system for live investigation of next generation botnets. In *Proc. of IEEE Conf. for Homeland Security, Cybersecurity Applications and Technology (CATCH '09)*, Mar. 2009.

[14] J. John, A. Moshchuk, S. Gribble, and A. Krishnamurthy. Studying spamming botnets using botlab. In *Proc. 6^{th} USENIX Symp. on Networked Systems Designs and Implementation (NSDI)*, Apr. 2009.

[15] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. Voelker, V. Paxson, and S. Savage. Spamalytics: an empirical analysis of spam marketing conversion. In *Proc. 15^{th} ACM Conf. Comp. & Comm. Security (CCS)*, Oct. 2008.

[16] C. Kanich, K. Levchenko, B. Enright, G. Voelker, and S. Savage. The Heisenbot uncertainty problem: Challenges in separating bots from chaff. In *Proc. 1^{st} USENIX Work. Large-Scale Exploits & Emergent Threats (LEET)*, Apr. 2008.

[17] L. Peterson and T. Roscoe. The design principles of PlanetLab. *ACM SIGOPS Operating Systems Review*, 40:11–16, Jan. 2006.

[18] M. A. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A multifaceted approach to understanding the botnet phenomenon. In *Proc. 6^{th} ACM SIGCOMM Conf. on Internet measurement (IMC)*, Oct. 2006.

[19] E. Ruitenbeek and W. Sanders. Modeling peer-to-peer botnets. In *Proc. 5^{th} Int. Conf. on Quantitative Evaluation of Systems (QuEST)*, pages 307–316, Sept. 2008.

[20] G. Sinclair, C. Nunnery, and B. Kang. The Waledac protocol: The how and why. In *Proc. 4^{th} Int. Conf. on Malicious and Unwanted Software (MALWARE)*, Oct. 2009.

[21] J. Stewart. Storm worm DDoS attack. `http://www.secureworks.com/research/threats/storm-worm`, Feb. 2007.

[22] B. Stock, J. Goebel, M. Engelberth, F. Freiling, and T. Holz. Walowdac analysis of a peer-to-peer botnet. In *Proc. Europ. Conf. Computer Network Defense (EC2ND)*, Nov. 2009.

[23] P. Wang, S. Sparks, and C. C. Zou. An advanced hybrid peer-to-peer botnet. In *Proc. 1^{st} Work. on Hot Topics in Understanding Botnets (HotBots)*, Apr. 2007.

[24] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of 5^{th} Symp. on Operating systems design and implementation (OSDI)*, pages 255–270, 2002.

[25] Y. Zhao, Y. Xie, F. Yu, Q. Ke, Y. Yu, Y. Chen, and E. Gillum. Botgraph: Large scale spamming botnet detection. In *Proc. 6^{th} USENIX Symp. on Networked Systems Designs and Implementation (NSDI)*, 2009.

Conficker and Beyond: A Large-Scale Empirical Study

Seungwon Shin
Success Lab, Texas A&M University
College Station, Texas, 77843, USA
seungwon.shin@neo.tamu.edu

Guofei Gu
Success Lab, Texas A&M University
College Station, Texas, 77843, USA
guofei@cse.tamu.edu

ABSTRACT

Conficker [26] is the most recent widespread, well-known worm/bot. According to several reports [16, 28], it has infected about 7 million to 15 million hosts and the victims are still increasing even now. In this paper, we analyze Conficker infections at a large scale, including about 25 millions victims, and study various interesting aspects about this state-of-the-art malware. By analyzing Conficker, we intend to understand current and new trends in malware propagation, which could be very helpful in predicting future malware trends and providing insights for future malware defense. We observe that Conficker has some very different victim distribution patterns compared to many previous generation worms/botnets, suggesting that new malware spreading models and defense strategies are likely needed. Furthermore, we intend to determine how well a reputation-based blacklisting approach can perform when faced with new malware threats such as Conficker. We cross-check several DNS blacklists and IP/AS reputation data from Dshield [6] and FIRE [7], and our evaluation shows that unlike a previous study [18] which shows that a blacklist-based approach can detect most bots, these reputation-based approaches did relatively poorly for Conficker. This raised the question, how can we improve and complement existing reputation-based techniques to prepare for future malware defense? Finally, we look into some insights for defenders. We show that neighborhood watch is a surprisingly effective approach in the Conficker case. This suggests that security alert sharing/correlation (particularly among neighborhood networks) could be a promising approach and play a more important role for future malware defense.

1. INTRODUCTION

Conficker worm (or bot) [26] first appeared in November 2008 and since then it has rapidly and widely spread in the world within a short period. It exploits a NetBIOS vulnerability in various Windows operating systems and utilizes many new, advanced techniques such as a domain generation algorithm, self-defense mechanisms, updating via Web and P2P, and efficient local propagation. As a result, it has infected millions of victims in the world and the number is still increasing even now [16, 28].

It is clear that the complex nature of Conficker makes it one of the state-of-the-art malware, and therefore the analysis of Conficker is very important in order to defend against it. A full understanding of Conficker can also help us in comprehending current and future malware trends. Existing research of Conficker analysis mainly falls into two categories. The first focuses on analyzing the Conficker binary and its behavior, revealing its malicious tricks such as the domain generation algorithm [23, 30]. In this direction, SRI researchers [23] and the Honeynet project [30] already provided excellent reports that analyzed Conficker in great detail. The second research category mainly focuses on analyzing the network telescope data [2] or DNS sinkhole data [13] to reveal the propagation pattern and victim distribution characteristics of Conficker on the Internet. There are very few studies in this direction, which is probably because it is very hard to obtain large scale real-world data of victims and the amount of data should be large enough to cover victims' global behavior. CAIDA [2] and Team Cymru [13] provided some initial reports which contain some very basic statistics on the scanning pattern and propagation information of Conficker. However, for a worm/bot that has infected so many victims and has so much potential to damage the Internet, it deserves a much deeper study. Such study is necessary because by analyzing this state-of-the-art botnet, we can gain more knowledge of current malware, e.g., how it differs from previous generation malware and whether such differences represent future trends or not. These deeper investigations could also provide new insights in developing new detection and defense mechanisms for current and future malware.

In this paper, we attempt to provide a deeper empirical measurement study of Conficker. We have collected a large-scale data set which contains almost 25 million Conficker victims with the help of *Shadowserver.org* (details on data collection are discussed in Section 3). We believe such scale is large enough to uncover Conficker's global patterns. We provide an extensive measurement of various distribution patterns of Conficker victims. Furthermore, we use a comparison- and cross-check-based methodology in our measurement study. We study the similarities and differences between Conficker and several other publicly reported worms/botnets. Then we analyze how these differences may affect existing reputation-based detection approaches. We

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

also investigate possible aspects that may be useful for Conficker and future malware defense.

In short, this paper makes the following contributions:

- We provide a large-scale empirical study of almost 25 million Conficker victims. By analyzing this data, we reveal many interesting aspects that were previously unknown and show that Conficker victims exhibit a very different distribution pattern from many previously reported botnets or worms. This difference could be a new trend or some ignored facts that are potentially important for future malware defense. Detailed information is in Section 4.

- We evaluate the effectiveness of existing reputation-based approaches for detecting emerging malware threats. They are considered as promising in defending against unknown malware compared to traditional signature-based approaches [1]. Through cross-checking several DNS blacklists and reputation data from Dshield [6] and FIRE [7], our evaluation shows that these reputation-based approaches are not effective for Conficker defense. This suggests that these reputation-based approaches need to be significantly improved and complemented by other techniques. Detailed information is in Section 5.

- We study the Conficker data and find that neighborhood watch is surprisingly effective to detect or predict new victims. This could suggest that alert sharing/correlation (among distributed collaborators, particularly neighborhood networks) could be an effective and promising technique to defend against future emerging threats and it needs more attention for such research. Detailed information is in Section 6.

2. RELATED WORK

Conficker binary analysis. Porras et al. from SRI International provided a very extensive study of the Conficker binary analysis [23]. They analyzed several variants of Conficker and revealed how Conficker propagates, how it infects others, how it evades anti-virus tools and how it updates itself. This provided very detailed and valuable information of Conficker behavior. The Honeynet project [30] also provides a detailed analysis of Conficker binary. These studies also provide scanning tools for detecting Conficker victims in the network.

Conficker data analysis. With the use of the telescope data, researchers from CAIDA provided a simple analysis on Conficker propagation [2]. The Telescope data mainly contains scanning traffic from Conficker victims, which reveals Conficker victim location and timing information to display how Conficker emerges and spreads on the Internet. However, such data is not complete due to the size limit of (passive) monitoring networks. Recently, researchers started to use the DNS sinkholing technique [13] to collect much more accurate Conficker victim data. A report from Team Cymru[13] analyzed the behavior of Conficker victims and provided some general distribution and propagation information. However, there is still a lack of some deep analysis of Conficker victims such as how different the victims are from previous malware. This paper is a first attempt to provide an empirical deep study of Conficker victims, reveal how they are distributed differently from previous generation malware, and how this affects current reputation-based defense mechanisms. In addition, we want to understand if there are some effective techniques for early detection of future variations of Conficker.

3. DATA COLLECTION

An interesting feature of Conficker is the resilient function of updating itself. To avoid detection, it automatically generates new domain names (of updating servers) [23, 30] and connects to those domain names to download an updated version of itself. This function greatly supports Conficker to increase the survivability and resilience. However, once the domain generation algorithm was cracked by researchers, it also provides a way to sinkhole and track the victims. By registering new domain names that will be used by Conficker victims on controlled servers, defenders can collect visits from hosts infected by Conficker. This approach is widely known as DNS sinkholing and has been successfully adopted by researchers that study Conficker [13].

With the aid of *Shadowserver.org*, we have collected the Conficker sinkhole data captured from January 1, 2010 to January 8, 2010. During this period, we observed 24,912,492 unique IP addresses of Conficker victims. We note that the accurate counting of worm/botnet victims is not an easy task because of the existence of DHCP, NAT, and many other issues [31, 25]. For example, Stone-Gross et al. [25] pointed out that there is a slight difference between the number of IP addresses and the number of real infected hosts. This is the limitation of almost all existing worm/botnet measurement studies. We do not intend to solve this problem in this paper. We simply report our observations from our collected data. Although the number may not be exact, with such a large scale it at least provides an estimation of overall characteristics and statistics of the Conficker botnet.

To obtain more interesting results, we surveyed previous work [15, 14, 19, 18, 31, 32, 24] about the behavior of nefarious worms and bots/botnets. They are used to compare with our Conficker result and to help us track whether infection trends have changed. Based on the information they provide, we selected seven measurement studies, which are summarized in Table 1. Of these, three are well-known network worms [15, 14, 19] and four are botnets [18, 31, 32, 24]. Note that some studies of botnets do not specify botnet names in their work, but they show the result of malicious nodes that send spam emails. Since most spam emails are delivered by botnets [18], we can reasonably assume that their studies represent the behavior of some bots or malware.

4. WHO IS WORKING FOR THE CONFICKER BOTNET?

In this section, we provide a basic but important network-level examination, which demonstrates fundamental characteristics of Conficker victims. We review how Conficker victims are distributed over the IP address space and ASes. Also, we investigate the bandwidth of Conficker victims and domain names that Conficker victims belong to. Finally, we survey portions of countries where Conficker victims heavily exist. Some of them are already provided by other studies [2, 13], but our work is more than just providing basic measurement results. To comprehend the radical alteration of

Malware [Work]	*Type*	*Data Source*	*Data Collection Time*
Botnet 1 [18]	Botnet	Sinkhole server	Aug. 2004 ∼ Jan. 2006
Botnet 2 [31]	Botnet	Hotmail	Jun. 2006 ∼ Sep. 2006
Botnet 3 [32]	Botnet	Spamhaus	Nov. 2006 ∼ Jun. 2007
Waledac [24]	Botnet	Infilatrion into Waledac	Aug. 2008 ∼ Sep. 2009
CodeRed [15]	Worm	Measurement	Jul. 2001 ∼ Oct.2001
Slammer [14]	Worm	Measurement	Jan. 2003
Witty [19]	Worm	Measurement	Mar. 2004

Table 1: Data source of previous worms/bots for comparison.

malware, we compare Conficker victims' network-level characteristics with those of previous well-known bots or worms.

4.1 Distribution Over Networks

We plotted each victim's IP address to determine how Conficker victims are distributed over the IP address space and found that they are not uniformly distributed in the whole IP address space; instead the distribution is highly biased, mostly concentrated in some specific ranges.

Result 1. (Distribution over the IP address space) *Most of hosts infected by Conficker are concentrated in several specific IP address ranges.*

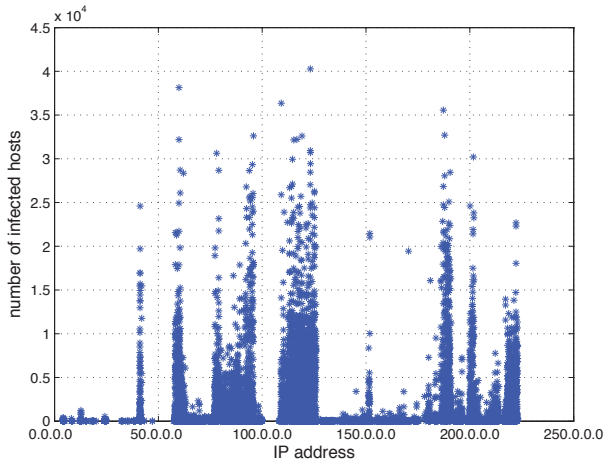


Figure 1: Distribution of infected hosts over IP address.

Figure 1 depicts the distribution of victims over the IP address space. The presence of several wide, sharp spikes, which represent densely infected areas, reveals that the victims are not uniformly distributed. Since the IP address ranges within these wide spikes could be regarded as more vulnerable, we inspected three notable wide spikes in detail. They are in the range of (109.* - 125.*), (77.* - 96.*), and (186.* - 222.*) and they cover around 87% of all victims. In particular, the widest and most prominent spike which is in the range of (109.* - 125.*), includes 9,303,423 infected hosts and accounts for 37.34% of the total number of Conficker victims. To get a more detailed view, we narrowed down the scope from the ranges to more specific networks. In the widest spike, we found that 123.* and 124.* networks are the main contributors. They comprise 1,701,438 infected hosts and account for 6.83% of all victims. We analyzed further and discovered that there are 40,278 Conficker victims in the 123.19.* network, which is around 61.9% of all possible IP addresses in that /16 subnet. Similar characteristics were observed in nearby networks such as the 123.22.* and the 123.23.*[1]

Result 1.1. (Distribution over IP address space - Comparison) *Some portions of IP address ranges were already affected by the previous botnets, but some ranges such as 109.* - 125.* are unique to Conficker.*

Comparing the distribution of Conficker victims over the IP address space with that of previous bots, we find that some ranges are similar to the previous results and some are unique to Conficker. The ranges of (77.* - 96.*) and (186.* - 222.*) are widely known as major locations of the Waledac bot [24]. Yet the interesting thing is that while the range of (109.* - 125.*) is one of the significant locations of Conficker, Waledac has no significant number of victims in that range. In addition, [18] investigated the IP address ranges of hosts infected by bots and they denoted that the ranges of (80.* - 90.*) and (210.* - 220.*) were major locations of bots, which is similar to Waledac analysis. However, both previous studies still did not point out the range of (109.* - 125.*) as a heavy contributor of bots. We tried to understand why the range of (109.* - 125.*) was not seen before. After investigating the data in this range, we concluded that the reason is most likely a change of infection trend, and we will elaborate on this in **Result 2.1**.

Since it is nearly impossible to monitor the *entire* Internet, it is more efficient to focus on specific (suspicious) networks that are more likely to contain commands directed by a botmaster. The IP address ranges within wide spikes, which are shown in Figure 1, can be good candidates that need to be focused.

Insight from Result 1 and 1.1 (Monitoring Networks more efficiently) *It is impossible to monitor all the IP addresses on the Internet, but we can monitor a limited number of specific ranges to efficiently detect commands and attacks in infected networks. Even though the ranges may be different for each botnet, there are still some common parts and they are good candidate ranges to monitor.*

[1]Since the 123.* network is in Class A network, it seems that there is no meaning in splitting it into subnetworks. However, people commonly split Class A networks into several /16 subnets to manage them efficiently. As in the case of 123.* network, we found that it is divided and assigned to several network providers. The 123.19.* network is one of them and it is assigned to *VietNam Post and Telecom Coorperation* and its *inetnum* is 123.19.0.0 - 123.19.255.255.

Representing identities of Conficker-infected hosts by IP address is often preferable in a way that it is precise and elaborate. However, the number of the infected IP addresses is so large that this makes it hard to grasp the global view of Conficker victims. Hence, we use the *Autonomous System (AS)*, which is a useful method for clustering hosts on the Internet for easier management and has been applied in previous measurement work, to group the hosts infected by Conficker.

Result 2. (Distribution over ASes) *Of all infected hosts, the top two ASes account for 28.37% of all victims and top 20 ASes cover 52.54% of all victims. In particular, most of the top rated ASes are located in Asia.*

Conficker victims are concentrated in a few ASes and most of the top infected ASes are located in Asia. As shown in Table 2, around 30% of infected hosts belong to one of only two ASes and more than 50% of infected hosts belong to one of the (top) 20 ASes. Most highly infected ASes are mainly distributed in Asia, particularly in China. This result also suggests that an approach to detect malicious hosts based on ASes would be practical.

ASN	*# Host*	*AS Name*	*Country*
4134	2825403	CHINA-BACKBONE	China
4837	1435411	CHINA169-BACKBONE	China
7738	385672	TELECOMUNICACOES	Brazil
3462	280957	HINET	Taiwan
45899	273577	VPNT-AS-VN	Vietnam
27699	260848	TELECOMUNICACOES	Brazil
9829	248444	BSNL-NIB	India
8167	237465	TELESC	Brazil
3269	231020	ASN-IBSNAZ	Italia
9121	207849	TTNET	Turkey
9394	195088	TELEFONICA	China
4812	182015	CRNET	China
4788	180876	CHINANET-SH-AP	Malaysia
8402	141130	TMNET-AS-AP	Russia
8151	138567	CORBINA-AS	Mexico
17974	137991	UNINET	Indonesia
4808	137672	TELKOMNET-AS2-AP	China
3352	135276	CHINA169-BJ	China
8708	128228	TELEFONICA-DATA-ESPANA	Romania
3320	126520	RDSNET	Germany

Table 2: Conficker victims in the top 20 ASes.

Result 2.1. (Distribution over ASes - Comparison) *Even though the top two ASes were also sources of previous botnets, most of other top rated ASes are newly emerged in the Conficker case.*

By comparing the result of the distribution over ASes with that of previous bots, we find that even if there are common ASes between Conficker and previous bots, there is a significant difference in the locations of infected ASes. Some studies [18, 31, 32] investigated which ASes are the major sources of the botnets that deliver spam emails[2]. We compare their findings with our result and denote it in Table 3. In [18], the authors analyzed data collected in 2004 - 2006 and pointed out that most of the bots are located in North America (particularly in USA), while in [31] and [32] in which data was collected in 2006 - 2007, it was emphasized that bots spread widely over the world. However, in the case of Conficker, ASes in the USA are no longer shown in the top 20 list. Instead, most highly infected ASes are located in Asia and South America.

From this result, we conclude that the trend of major locations of bot infected hosts is still changing; *(i) mainly located in North America*, *(ii) widely spread over the World*, *(iii) popular in Asia and South America*. This trend guides us to observe Asia and South America more closely than North America, which used to be the major source of spam email when we built blacklists to prevent spam at the time. It is important that the trend of major sources of bots is changing. Also, we find that four ASes in Conficker are never seen in previous results. Two of them are in Asia (Vietnam and India) and two of them are in South America (Brazil).

Insight from Result 2 and 2.1. (Change of Infection Trend) *North America used to be the main contributors of botnets, but now Asia and South America contribute more. This means that the locations of the main sources of botnets are changing and we may chase this trend (e.g., new malware spreading models and defense strategies are probably needed).*

4.2 Distribution Over Domain Names

In this section, we inspect the domain names of each victim using DNS reverse lookup.[3] A domain name indicates a group in which a host belongs and it can be a good way to reveal the host itself because domain names are expressed in easy and comprehensible words.

Result 3. (Distribution over Domain Name) *The .br, .net and .cn domains cover around 24.42% of Conficker victims. Interestingly, one of the third level domains covers around 7% of infected hosts, which means it contains more than 1,700,000 victims.*

As shown in Table 4, only a few domains account for about 20% of hosts infected by Conficker. This does not solely apply to top level domains but to all second level domains and third level domains as well. In the case of top and second level domain names, their scope is quite broad and it is hard to find any big advantage when compared to IP address range or AS number. However, for third level domain names, it is possible to focus on small sets of victims. It is useful to monitor victims because the top third level domain includes numerous Conficker victims. In particular, we find that domain *163data.com.cn* accounts for 6.88% of infected hosts. Also, more than 99% of victims in *163data.com.cn* include the word *dynamic* in their fourth level domain names. From this, we can guess that they are using dynamic IP addresses, as their names imply. This result is similar to [31] which uncovers dynamic IP addresses as a main source of most spam emails.

[2]In [32], they only present the top five of ASes, and that is why we could not compare the whole list.

[3]In our DNS reverse lookups, about 49% of victims did not return valid results and therefore we labeled them as "Unknown", shown in Table 4. Since previous studies also showed similar rates of "unknown" domains, we leave them in the table.

Conficker		Botnet 1 [18]		Botnet 2 [31]		Botnet 3 [32]	
ASN	*Country*	*ASN*	*Country*	*ASN*	*Country*	*ASN*	*Country*
4134	China	766	Korea	4134	China	4766	Korea
4837	China	4134	China	4837	China	19262	USA
7738	Brazil	1239	USA	4776	Australia	3215	France
3462	Taiwan	4837	China	27699	Brazil	4837	China
45899	Vietnam	9318	Japan	3352	Spain	4134	China
27699	Brazil	32311	USA	5617	Poland	no info.	no info.
9829	India	5617	Poland	19262	USA	no info.	no info.
8167	Brazil	6478	USA	3462	Taiwan	no info.	no info.
3269	Italia	19262	USA	3269	Italy	no info.	no info.
9121	Turkey	8075	USA	9121	Turkey	no info.	no info.

Table 3: Top 10 ASes hosting Conficker and Spamming Botnets.

Top Level	*Percentage*	*Second Level*	*Percentage*	*Third Level*	*Percentage*
Unknown	48.81%	Unknown	48.81%	Unknown	48.81%
br	8.83%	com.cn	6.89%	163data.com.cn	6.88%
net	8.65%	net.br	4.61%	veloxzone.com.br	1.96%
cn	6.94%	com.br	4.20%	dynamic.hinet.net	1.86%
ru	5.01%	hinet.net	1.91%	telesp.net.br	1.69%
it	2.36%	telecomitalia.it	1.55%	retail.telecomitalia.it	1.46%
ar	1.54%	corbina.ru	0.99%	brasiltelecom.net.br	1.39%
in	1.35%	ny.adsl	0.93%	broadband.corbina.ru	0.99%
com	1.21%	com.mx	0.90%	kd.ny.adsl	0.93%
mx	1.16%	com.ar	0.84%	prod-infinitum.com.mx	0.85%

Table 4: Top 10 Domain Names hosting Conficker Victims in each level.

Result 3.1. (Distribution over Domain Name - Comparison) *The .net domain is still prevalent, but new domains such as .br, .cn, and .ru have recently emerged as heavy resources of botnets. The .com and .edu domains used to be the major sources of worms, but now they cast off the yoke of malicious domains.*

Comparing the domain result with previous work, we found that a few domains that were not previously seen in Conficker. Also, we found that *.com* and *.edu* domains, which used to be nefarious domains, are now relatively clean. Unfortunately, because the previous work does not show second level and third level domain distributions, we could only compare top level domains. In previous studies, top contributors of infected domains are *.net, .com* and *.edu*. However, in the case of Conficker, things have changed. While the *.net* domain is still prevalent, there are newly emerged domains which are not shown in the previous work: *.cn, .ru, .in, and .mx*. All domains that are newly seen represent their countries and we call these ccTLDs (Country Code Top Level Domains). The report from Verisign [29] shows that the registration rate of above ccTLDs has increased explosively for the past three years. This implies that the number of hosts in newly registered domains have increased exponentially. Therefore we may monitor more closely whether they are infected by malware or not, since they may not be on any blacklists. The more interesting part is *.edu* and *.com* domains are no longer serious sources of malware. Of course, there are infected hosts which still belong to those domains, but its coverage is reduced to 1.21% in *.com* and 0.0096% in *.edu*. This result implies that the networks in *.com* and *.edu* domains are probably better managed and protected than before. The comparison result is summarized in Table 5.

Result 3.2. (Distribution over Domain Name - Sensitive Domain Name) *There are Conficker victims in government networks and companies listed in Fortune 100, even though the number of infected hosts is small.*

Besides sending DDoS packets and spam emails, a botnet can steal sensitive information from victims [11]. If hosts infected by a bot belong to critical networks such as government and military networks that contain sensitive information, a botmaster can steal important information from them. Using our Conficker data, we investigated how many victims are affiliated with government or military networks and we found 714 such victims. Surprisingly, victims in government networks are not limited to a few countries, instead they are spread around 70 countries including U.S.A., Parkistan, India and China. Also, we investigated how many victims are in well-known companies. To do this, we used the *Fortune 100 Company List* [8] and we found 2,847 such hosts. Conficker victims still exist within several reputable companies such as HP and IBM.

Insight from Result 3, 3.1 and 3.2. (Watch out for new and sensitive Domains!) *It is nearly impossible to monitor all domain names. However, we have observed that newly registered domains are more vulnerable and more easily infected by Conficker. Hence, it is necessary to closely monitor those recently registered domains. In addition, even though the number of victims is not large, a botmaster of Conficker can steal sensitive information from government and top rated company networks.*

4.3 Distribution over Bandwidth

Besides IP address, AS and domain names, bandwidth gives us information that shows us what kinds of networks Conficker victims belong to. It also helps to predict the power of the botnet. For instance, if we know there are one

Conficker		CodeRed		Slammer		Witty	
Top level	*Percentage*	*Top level*	*Percentage*	*Top level*	*Percentage*	*Top level*	*Percentage*
Unknown	48.81%	Unknown	47.22%	Unknown	59.49%	net	33%
br	8.83%	net	18.79%	net	14.37%	com	20%
net	8.65%	com	14.41%	com	10.75%	Unknown	15%
cn	6.94%	edu	2.37%	edu	2.79%	fr	3%
ru	5.01%	tw	1.99%	tw	1.29%	ca	2%
it	2.36%	jp	1.33%	au	0.71%	jp	2%
ar	1.54%	ca	1.11%	ca	0.71%	au	2%
in	1.35%	it	0.86%	jp	0.65%	edu	1%
com	1.21%	fr	0.75%	br	0.57%	nl	1%
mx	1.16%	nl	0.73%	uk	0.57%	ar	1%

Table 5: Top 10 Domain Names hosting Conficker, Codered, Slammer and Witty.

million Conficker victims in the world and most Conficker victims are in networks with bandwidth less than 1 Kbps, we deduce that it could generate 1 Gbps traffic in the best case. To measure the bandwidth, we use *Tmetric* [27] which sends ICMP packets to the target network and provides a measured bandwidth result. Since *Tmetric* needs to contact the target network to estimate the bandwidth, we can not get the bandwidth result without live target networks and hosts. It takes quite a long time to contact each host and measure the bandwidth, so we only contact one host in the subnetworks (/24) where Conficker victims exist. We reasonably assume that hosts in the same subnetwork (/24) have the same bandwidth.

Result 4. (Bandwidth Distribution) *About 99% of Conficker victims have bandwidth less than 1 Mbps and this means that most of them are ADSL or Modem/Dialup users.*

We find that most victims are using Modem/Dialup or ADSL networks. As shown in Figure 2 (a), about 90% of Conficker victims are in the network whose bandwidth is less than 200 Kbps and around 99% of victims are residing in the network whose bandwidth is less than 1 Mbps. This result is similar to [10] and [31] which denote most bots are using ADSL or Dialup networks. When we conducted this measurement, we found interesting patterns between the bandwidth of a subnet and the number of infected hosts in the subnet.

Result 4.1. (Bandwidth Distribution - relation with the numbers of victims) *The networks that have low bandwidth are likely to have more Conficker victims than those with high bandwidth.*

We suspect that there is a relationship between the bandwidth of a network and the number of infected hosts of the network. As shown in Figure 2 (b), the bandwidth of the subnet is inversely related to the number of infected hosts in the subnet. We think that this pattern is related to the manageability of each network. A network with high bandwidth indicates consuming high setup cost and it also means the network is that worthy. And we could infer that such worthy network is under reasonably good maintenance.

Insight from Result 4 and 4.1. (Examine ADSL or Modem/Dialup networks) *Hosts with ADSL or Modem/Dialup connections are still very vulnerable.*

4.4 Distribution over Geographic Location

Result 5. (Geographic Location) *34.47% of infected hosts are located in China, which is larger than the total number of Conficker victims from the next top eight countries.*

As shown in Table 6 on the distribution over countries, the top ten countries include over 70% of Conficker victims, China ranks number one by a large margin. Conficker victims are distributed over most of the world including Asia, Europe, and South America, but interestingly, only 1.1% of victims are located in North America. This result is somewhat different from previous infection patterns.

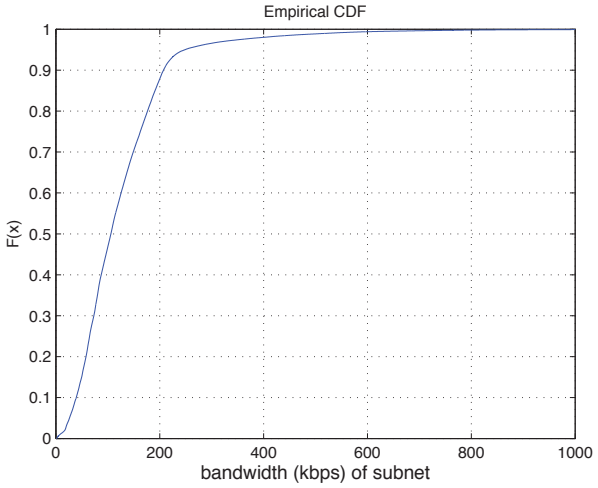
Result 5.1. (Geographic Location - Comparison) *In previous worms and botnets, most the infected hosts were located in North America - especially in USA, but in Conficker, most victims are located in the Asian region - especially in China.*

We compare the country distribution with that of other worms and bots to determine whether it is different or similar and we find that the location of heavy malware contributors is changing. Even though we could not get the exact country distribution from the previous work [18] [31], we are able to estimate which country had more victims based on their distribution over ASes. From Table 6 and 3, we observe that worms prevalent several years ago were mainly located in North America. In previous botnets, [31] and [32] show that victims are mainly located in both Asia and North America, but [18] and [24] denote that most victims are located in North America. However, contrast to the results of previous work, we find that Conficker victims are mainly located in Asia and not in North America, where only 1.1% of victims are located. Therefore, changing monitoring focus from North America to Asia seems reasonable.

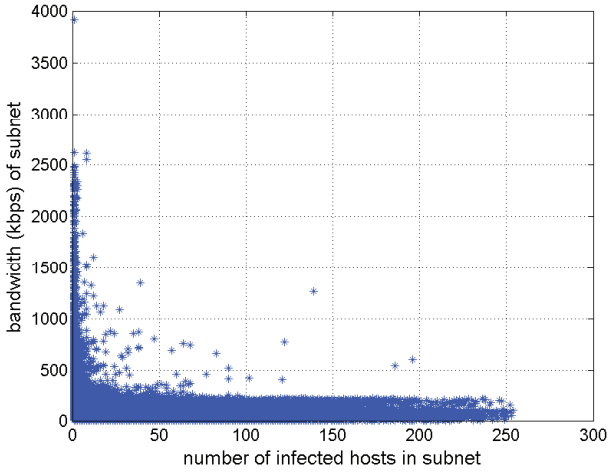
Insight from Result 5 and 5.1. (From North America to Asia - Confirmed) *We clearly observe that the hosts infected by Conficker are mainly located in Asia and not in North America, as also shown in Result 2 and 2.1.*

5. HOW WELL DO REPUTATION-BASED DETECTION SYSTEMS DETECT CONFICKER?

In this section, we examine how well current reputation-



(a)



(b)

Figure 2: Bandwidth measurement of Conficker victims.

Conficker		Waledac		CodeRed		Slammer		Witty	
Country	*%*	*Country*	*%*	*Country*	*%*	*Country*	*%*	*Country*	*%*
China	34.47%	USA	17.34%	USA	43.91%	USA	42.87%	USA	26.28%
Brazil	9.43%	U.K	7.76%	Korea	10.57%	Korea	11.82%	U.K	7.27%
Russia	7.39%	France	7.04%	China	5.05 %	Unknown	6.96%	Canada	3.46 %
India	4.45%	Spain	5.90%	Taiwan	4.21%	China	6.29%	China	3.36%
Italy	3.56%	India	5.50%	Canada	3.47%	Taiwan	3.98%	France	2.94%
Vietnam	2.81%	no info.	no info.	U.K.	3.32%	Canada	2.88%	Japan	2.17%
Taiwan	2.59%	no info.	no info.	Germany	3.28%	Australia	2.38%	Australia	1.83%
Germany	2.03%	no info.	no info.	Australia	2.39%	U.K.	2.02%	Germany	1.82%
Argentina	2.00%	no info.	no info.	Japan	2.31%	Japan	1.72%	Netherlands	1.36%
Indonesia	1.85%	no info.	no info.	Netherlands	2.16%	Netherlands	1.53%	Korea	1.21%

Table 6: Top 10 countries where Conficker, Waledac, Codered, and Slammer are located.

based detection systems detect Conficker. A DNS blacklist is an effective approach to detect malicious hosts and networks based on reputation [1]. We investigate how well it detects Conficker victims to verify its effectiveness. Also, we examine other reputation-based detection systems such as Dshield [6] and FIRE [7] to check if they could successfully detect Conficker victims.

5.1 DNS Blacklist

We have investigated several well-known blacklists such as DNSBL [5], SORBS [20], SpamHaus [22], and SpamCop [21] to see how many victims of Conficker are on their blacklists. We tested all 24,912,492 infected hosts and we found out that only 4,281,069 hosts are on blacklists which is only 17.18% of all victims.

Result 6. (DNS Blacklist) *DNS blacklists only cover a small portion of Conficker victims. More specifically, only 17.18% of Conficker victims are found on any of four DNS blacklists.*

Our investigation result is quite different from the previous work [18] which shows about 80% of bot infected hosts are already on some blacklists and we believe that the disparity is caused by the difference of distribution of infected hosts. As we mentioned in Section 4.1 and 4.2, the distribution of Conficker victims (over IP address space, ASes, Domain names and Countries) is different from the previous work, and this makes it hard to build effective blacklists for detecting emerging malicious hosts/networks, because blacklists highly depend on the reputation of hosts and networks obtained from their previous records (and currently heavily rely on spam activity records).

Insight from Result 6. (Unfortunately, blacklists can not help us all the time) *Only less than 20% of victims are on DNS blacklists, which means that we need better ways to detect future emerging malware.*

5.2 Dshield and FIRE

Some other reputation-based detection systems are also provided to complement DNS blacklists, and we need to investigate their performance of detection. Since most DNS blacklists are mainly to detect hosts or ASes sending spam, they may not detect other malicious behaviors (potentially) performed by (emerging) infected hosts. There are several studies that try to detect network scanning attacks or web-based attacks and Dshield [6] and FIRE [7] are good examples of them. Dshield provides information to detect hosts or ASes sending suspicious network scanning/attacking packets, and FIRE [7] lists malicious ASes which frequently host

rogue networks by measuring their reputation. We plan to inspect how many Conficker victims are notified by Dshield and FIRE.

Result 7. (Dshield) *Only 0.33% of victims of Conficker are found on the list of malicious IP addresses reported by DShield, and most of the top ASes infected by Conficker are not on the malicious AS list of Dshield.*

Checking Conficker victims against the list provided by Dshield [4], we found that only a small portion of hosts and ASes are on the list. We investigated 588,797 IP addresses presented by Dshield, and they denoted world-wide attackers/scanners that were detected by all kinds of IDSs and reported to DShield. Since one of the infection vectors in Conficker is random IP scanning [17], we expect a large portion of Conficker victims to show up in Dshield. However, we only find 82,856 hosts from the list. This shows that these Conficker victim hosts are probably easy targets of many previous malware. However, DShield is still not good at catching major portions of new emerging malware such as Conficker. Similarly, we examined the malicious AS list provided by Dshield and we only observed 83 Conficker infected ASes out of 10,584 ASes given by Dshield. Only one of them (AS4812) is a serious contributor of Conficker (ranked 12th among infected ASes) but the rest are not as critical as *AS4812*. Most of them cover less than 0.02% of Conficker victims.

Result 8. (FIRE) *Most highly infected ASes by Conficker are not reported by FIRE.*

We compared our infection list of ASes with the results provided by FIRE as well and we want to know whether FIRE is helpful in detecting Conficker victims. Although FIRE denotes *AS4134* as the 8th most malicious AS in its list, most of other heavily infected ASes by Conficker are not shown in the top 500 malicious ASes of FIRE. Some of the main contributing ASes to Conficker have never shown up on FIRE's list.

Insight from Result 7 and 8. (New and complementary detection approaches are needed) *DNS blacklists, Dshield and FIRE detect only a small portion of Conficker victims. This means that these reputation-based approaches are not the perfect solution. We need to improve them significantly and complement them with other approaches.*

When we tested Dshield and FIRE, we expected that they could complement DNS blacklists, but the result is not very positive. This implies that these reputation-based systems alone are far from enough to protect the Internet from emerging threats. We believe that new detection systems based on anomalous behaviors of malware could be a good complementary approach to them.

6. CAN NEIGHBORHOOD WATCH HELP?

Conficker still uses network scanning to infect other hosts on the Internet as previous worms and bots did, and it also adopts several advanced skills to infect hosts efficiently. The spreading techniques of Conficker can be classified into two categories [3, 17]; *(i) infecting random hosts* and *(ii) infecting nearby hosts*. Conficker has a function of scanning randomly selected IP addresses. Although this will help Conficker to spread globally, it is not probably very efficient these days because most networks are protected by firewalls or Network Intrusion Detection/Prevention Systems. To propagate more efficiently, Conficker adopts several interesting techniques to infect hosts nearby; (1) an ability to infect other hosts in the same subnet, (2) an ability to infect hosts in the nearby subnets, and (3) an ability to infect portable storage devices.

The diverse infection techniques of Conficker lead us to ask this question: *"Which vector is more effective to infect hosts?"*. Some previous studies suggested that second approach - *(ii) infecting nearby hosts* - is probably more dominant in the Conficker case [17, 12]. We think that this seems reasonable, because even though most networks are protected well from outside threats, they are still open to internal attacks. However, they do not show concrete evidence to support it.

To determine whether this hypothesis is correct, we constructed a test. Prior to explaining our test, we declare that we will use /24 subnet as a basic unit in our test. And we make the following definition to simplify the test. We define two terms: *(i) "camp" is the group of /24 subnets whose /16 subnet is the same and locations are close together, and (ii) each /24 subnet is a "neighbor" of nearby /24 subnets in the same camp.* Sometimes, even if two /24 subnets are in the same /16 subnet, their physical locations could be far from each other. However, since our concept of *"camp"* is each /24 subnet with both nearby IP address and physical location, we should consider its location as well. Based on the above definition, we establish a hypothesis as follows. *Of the two infection vectors of Conficker, suppose the second infection vector plays a dominant role, the infection pattern*[4] *of a /24 subnet will be similar to that of its "neighbors" in the same "camp".* In other words, the hosts in nearby networks of infected host are more likely to be selected as future victims than randomly chosen hosts.

To evaluate this hypothesis, we have tested the following scenarios. First, we divide hosts into /24 subnets and assign each /24 subnet into a "camp" based on our definition. Second, we investigate the infection pattern of each /24 subnet to see whether the infection pattern of each /24 subnet is similar to its "neighbors". We use *Variance-Mean Ratio (VMR)* [9] for a numerical expression. In this test, we measure the mean and variance value of the numbers of infected hosts of each /24 subnet in each "camp", and calculate *VMR* for each "camp". If the value of *VMR* is less than one, distribution of the data set shows under-dispersion with mean value in the center, which means that infection patterns of /24 subnets in the "camp" are very similar to each other.

Result 9. (Neighborhood) *Most /24 subnets show similar infection patterns (numbers of infected hosts) with their "neighbors". The closer they are located with each other, the more similar in their infection patterns.*

We measured the *VMR* value of each "camp" and we found that more than 70% of "camps" denoted that their /24 subnet members are similar to each other. From this result, we reasonably infer that the dominant infection vector of Con-

[4]We use the number of infected hosts of /24 subnet as a feature to represent an infection pattern.

Within Distance	*# of all "camps"*	*# of "camps" whose /24 subnet members are similar to each other*
≈ 100km	85,246	62,121 (72.87%)
≈ 200km	65,748	44,633 (67.88%)
≈ 300km	54,415	36,495 (67.06%)

Table 7: The number of all "camps" and "camps" whose members are similar to each other.

ficker is to infect nearby hosts. The test result is shown in Table 7. When we did this test, we got three types of "camps" based on its geographical information. For instance, if we set the distance metric for the "camp" as 100km which means that all /24 subnets in the "camp" have the same /16 subnet and they are within 100km of each other, we found 85,246 "camps" from our data and we discovered 62,121 "camps" whose /24 subnet members are similar to each other. We observed that more than 67% of "camps" showed that their /24 subnet members are similar to each other. The closer their locations are, the clearer this pattern is shown. This result tells us that Conficker is more likely to select nearby hosts than randomly chosen hosts and this means Conficker victims are mainly infected by neighbor networks/hosts. We deduce from this result that infection from the inside could be more harmful than the threats from the outside. Usually, most enterprise networks and ISPs protect their internal hosts using firewalls and IPS/IDS from external attacks, but there are very few approaches to protect hosts from internal threats.

Result 9.1 (Detection based on neighborhood information) *We could detect unknown victims by sharing and correlating neighbor alert information, even if we only know small sets of families and its neighbors.*

Based on previous results, we propose an approach of detecting (or early warning) emerging (unknown) infected /24 subnets using neighborhood information and we show that the approach can detect unknown infected /24 subnets with more than 90% of accuracy. From the above test, we find that Conficker victims share their infection patterns with their neighbors, and this finding gives us an intuition that collecting and sharing neighborhood information would be helpful to detect unknown malware or provide early warnings. To validate this intuition, we have tested the simple scenario of *"We only have small portions of information of benign and malicious hosts, but we can gather neighborhood information. Then, how many unknown malicious hosts can we detect (or predict) based on neighborhood information?"*.

As a method of considering neighborhood information, we use the K-Nearest Neighbor (KNN) classification algorithm, because it is a very popular approach that classifies unknown examples using the most similar "neighbors" in the known examples. When we apply the KNN algorithm to our data, we need the following preparations.

- **define classes:** *in this test, we define two classes; benign (normal /24 subnet) and malicious (/24 subnet which has Conficker victims)*
- **collect data:** *we use our Conficker data for malicious data, and we collected the same number of benign /24 subnets as malicious /24 subnets.*[5]
- **divide data:** *we randomly select 20% of data from both data sets for training samples and other 80% of data is used for testing.*

After all preparation was completed, we used the KNN algorithm (we use 3 for K and use IP address to calculate the distance) to our data and found that it can detect unknown infected /24 subnets with a high accuracy. As shown in Table 8, we find that even if we only know a small part of Conficker data (20%), we can still predict other infected /24 subnets within more than 90% accuracy with reasonable True Positive (TP) and False Positive (FP)[6] rates. This detection result implies that if we share neighbor information, we could detect unknown victims or provide early warnings more efficiently.

Detection Accuracy	*TP rate*	*FP rate*
91.59%	91.65%	8.5%

Table 8: Accuracy, TP and FP rate of the Detection Approach based on Neighborhood Information.

Insight from Result 9 and 9.1. (Neighborhood watch) *We observe that a large portion of victims could be infected by nearby victims and find that it is very important to share threat information with neighborhood networks. And this insight implies that further research is needed for developing new detection/defending approaches based on cooperated/shared (alert) information (and probably in an efficient privacy-preserving way).*

7. CONCLUSION

In this paper, we have studied a large-scale Conficker infection data to discover (i) their distribution over networks, ASes and etc, (ii) difference from previous bots/worms (iii) the effectiveness of current reputation-based malware detection/warning systems, and (iv) some insight to help detect future malware.

Our analysis of Conficker victims and cross-comparison results allowed us to obtain profound insights of Conficker victims. They also guide us to understand the trends of malware infections and to find interesting ideas that can aid the design of future malware detecting systems. We revealed that current reputation-based malware detecting systems depending on previously known information are not enough to detect most Conficker victims. This result suggests that different kinds of (complementary) detection systems such as an anomaly-based detection system are needed.

[5]As a result, we have 1,300,000 malicious /24 subnets (infected by Conficker), and 1,300,000 benign /24 subnets (NOT infected by Conficker or other malware).

[6]TP denotes the rates that the detector classifies real malicious networks correctly, and FP denotes the rates that the detector classifies benign networks as malicious.

We provide a basis that proves the hypothesis of *"A Conficker bot is more likely to infect nearby hosts than randomly chosen hosts"* and we believe that it calls for more research of detection systems which are based on watching/sharing/correlating neighborhood information.

Acknowledgments

We greatly thank Chris Lee and Shadowserver.org for providing the data used in this paper. We also would like to thank our shepherd, Sven Dietrich, and anonymous reviewers for their insightful comments and feedback to improve the paper. This material is based upon work supported in part by the Office of Naval Research under Grant no. N00014-09-1-0776, the National Science Foundation under Grant CNS-0954096, and the Texas Higher Education Coordinating Board under NHARP Grant no. 01909. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Office of Naval Research, the National Science Foundation, and the Texas Higher Education Coordinating Board.

8. REFERENCES

[1] M. Antonakakis, R. Perdisci, D. Dagon, W. Lee, and N. Feamster. Building a Dynamic Reputation System for DNS. In *Proceedings of USENIX Security of Symposium*, Aug. 2010.

[2] CAIDA. Conficker/Conflicker/Downadup as seen from the UCSD Network Telescope. `http://www.caida.org/research/security/ms08-067/conficker.xml`.

[3] E. Chien. Downadup: Attempts at Smart Network Scanning. `http://www.symantec.com/connect/blogs/downadup-attempts-smart-network-scanning`.

[4] DHIELD. All suspicious Source IPs in DSHIELD. `http://www.dshield.org/feeds/daily_sources`.

[5] DNSBL. invaluement DNSBL (an anti-spam blacklist). `http://dnsbl.invaluement.com/`.

[6] DSHIELD. Cooperative Network Security Community. `http://www.dshield.org/`.

[7] FIRE. Finding Rogue Networks. `http://maliciousnetworks.org/`.

[8] Fortune. Fortune 100 companies. `http://money.cnn.com/magazines/fortune/`.

[9] U. G. and C. I. *Oxford Dictionary of Statistics (2nd edition)*. Oxford University Press, 2006.

[10] T. Holz, C. Gorecki, and F. Freiling. Detection and Mitigation of Fast-Flux Service Networks. In *Proceedings of NDSS Symposium*, Feb. 2008.

[11] N. Ianelli and A. Hackworth. Botnets as a Vehicle for Online Crime. 2005.

[12] S. Krishnan and Y. Kim. Passive identification of Conficker nodes on the Internet. In *University of Minnesota - Technical Document*, 2009.

[13] J. Kristoff. Experiences with Conficker C Sinkhole Operation and Analysis. In *Proceedings of Australian Computer Emergency Response Team Conference*, May 2009.

[14] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer Worm. In *Proceedings of IEEE Security and Privacy*, May 2003.

[15] D. Moore, C. Shannon, and K. Calffy. Code-red: a case study on the spread and victims of an internet worm. In *Proceedings of ACM SIGCOMM Workshop on Internet Measurement*, Nov. 2002.

[16] B. N. Online. Clock ticking on worm code. `http://news.bbc.co.uk/2/hi/technology/7832652.stm`.

[17] P. Porras, H. Saidi, and V. Yegneswaran. A Foray into Conficker's Logic and Rendezvous Points. In *Proceedings of USENIX LEET*, Apr. 2009.

[18] A. Ramachandran and N. Feamster. Understanding the Network-Level Behavior of Spammers. In *Proceedings of ACM SIGCOMM*, Sep. 2006.

[19] C. Shannon and D. Moore. The Spread of the Witty Worm. In *Proceedings of IEEE Security and Privacy*, May 2004.

[20] SORBS. Fighting spam by finding and listing Exploitable Servers. `http://www.au.sorbs.net/`.

[21] SPAMHAUS. Spamcop.net. `http://www.spamcop.net/`.

[22] SPAMHAUS. The SPAMHAUS Project. `http://www.spamhaus.org/`.

[23] SRI-International. An analysis of Conficker C. `http://mtc.sri.com/Conficker/addendumC/`.

[24] B. Stock, M. E. Jan Goebel, F. C. Freiling, and T. Holz. Walowdac Analysis of a Peer-to-Peer Botnet. In *Proceedings of European Conference on Computer Network Defense (EC2ND)*, Nov. 2009.

[25] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your Botnet is My Botnet: Analysis of a Botnet Takeover. In *Proceedings of ACM CCS*, Nov. 2009.

[26] M. S. Techcenter. Conficker worm. `http://technet.microsoft.com/en-us/security/dd452420.aspx`.

[27] Tmetric. Bandwidth Measurement Tool. `http://mbacarella.blogspot.com/projects/tmetric/`.

[28] UPI. Virus strikes 15 million PCs. `http://www.upi.com/Top_News/2009/01/26/Virus-strikes-15-million-PCs/UPI-19421232924206/`.

[29] Verisign. The Domain Name Industry Brief. `http://www.verisign.com/domain-name-services/domain-information-center/domain-name-resources/domain-name-report-sept09.pdf`.

[30] D. Watson. Know Your Enemy: Containing Conficker. `http://www.honeynet.org/papers/conficker`.

[31] Y. Xie, F. Yu, K. Achan, E. Gillum, M. Goldzmidt, and T. Wobber. How Dynamic are IP Addresses? In *Proceedings of ACM SIGCOMM*, Aug. 2007.

[32] Y. Xie, F. Yu, K. Achan, R. Panigraphy, G. Hulte, and I. Osipkov. Spamming Botnets: Signatures and Characteristics. In *Proceedings of ACM SIGCOMM*, Aug. 2008.

Spam Mitigation using Spatio-Temporal Reputations from Blacklist History *

Andrew G. West, Adam J. Aviv, Jian Chang, and Insup Lee
Dept. of Computer and Information Science - University of Pennsylvania - Philadelphia, PA
{westand, aviv, jianchan, lee}@cis.upenn.edu

ABSTRACT

IP blacklists are a spam filtering tool employed by a large number of email providers. Centrally maintained and well regarded, blacklists can filter 80+% of spam without having to perform computationally expensive content-based filtering. However, spammers can vary which hosts send spam (often in intelligent ways), and as a result, some percentage of spamming IPs are not actively listed on any blacklist. Blacklists also provide a previously untapped resource of rich historical information. Leveraging this history in combination with spatial reasoning, this paper presents a novel reputation model (PreSTA), designed to aid in spam classification. In simulation on arriving email at a large university mail system, PreSTA is capable of classifying up to 50% of spam not identified by blacklists alone, and 93% of spam on average (when used in combination with blacklists). Further, the system is consistent in maintaining this blockage-rate even during periods of decreased blacklist performance. PreSTA is scalable and can classify over 500,000 emails an hour. Such a system can be implemented as a complementary blacklist service or used as a first-level filter or prioritization mechanism on an email server.

1. INTRODUCTION

Roughly 90% of the total volume of email on the Internet is considered spam [5], and IP-based blacklisting has become a standard tool in fighting such influxes. Spammers often control large collections of compromised machines, *botnets*, and vary which hosts act as the spamming mail servers. As a result, some 20% of spam emails received at a large spam trap in 2006 were not listed on any blacklist [21].

Blacklists provide only a static view of the current (or recently active) spamming IP addresses. However, when viewed over time, blacklists provide dense historical (temporal) information. Upon inspection, interesting properties emerge; for example, more than 25% of the IPs once listed on the blacklist were re-listed within 10 days, and overall, 45% were re-listed during the observation period.

It is known that spamming IP addresses exhibit interesting spatial properties. Previous studies have shown that spamming IPs are distributed non-uniformly throughout the address space [19, 21, 28], and they can often be clustered into spatial groups indicative of spamming behavior. For example, AS-membership has been shown to be a strong predictor of spamming likelihood [11], as well as BGP prefixes, and the host-names of reverse DNS look-ups [19].

In this paper we propose a novel method to combine blacklist histories with spatial context to produce predictive reputation values capable of classifying spam. Our model, **Pre**ventive **S**patio-**T**emporal **A**ggregation (PreSTA), monitors blacklist dynamics, interpreting listings as a record of negative feedback. An entity (*i.e.*, an IP address) is then evaluated based on its own history of negative feedback and the histories of spatially related entities. Spatial adjacency is multi-tiered and defined based on multiple grouping functions (*e.g.*, AS-membership, subnet, *etc.*). A reputation value is computed for each grouping, and these are combined using a standard machine learning technique to produce ham/spam classifications.

We implemented PreSTA and analyzed incoming email traces at a large university mail server. We found that PreSTA can classify an additional 50% of spam not identified by blacklists alone while maintaining similar false-positive rates. Moreover, when PreSTA is used in combination with traditional blacklists, on average 93% of spam is *consistently* identified without the need for content-based analysis. This result was found to be stable: As the underlying blacklist suffers large deviations in detection accuracy, PreSTA maintains steady-state performance. Further, PreSTA is highly *scalable*: Over 500,000 emails an hour can be scored using a single-threaded implementation on a commodity server.

We do not propose that PreSTA can (or should) replace context-based filtering. Instead, PreSTA can be leveraged just as blacklists are today – as a preliminary filter to avoid more computationally expensive analysis. Use-cases could include a complimentary service to blacklists (perhaps implemented by the blacklist provider) or an email prioritization mechanism for overloaded mail servers.

PreSTA's applicability is not confined to email spam detection. Related work has already shown PreSTA reputations helpful in prioritizing edits and detecting vandalism on Wikipedia [30], and PreSTA may be further applicable to an entire class of dynamic trust management problems [9,

*This research was supported in part by ONR MURI N00014-07-1-0907. POC: Insup Lee, lee@cis.upenn.edu

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

29] that are characterized by the need for decision-making in the presence of uncertainty and partial-information.

2. RELATED WORK

Spam filtering based on network-level properties of the source IP address is a popular choice for mitigating spam. Unlike content-based filters (*e.g.*, those based on Bayesian quantifiers [24]), these techniques tend to be computationally inexpensive while achieving relatively good performance.

IP blacklists [3, 7] are one such network-level filtering strategy. Blacklists are collections of known spamming IP addresses collated from various institutions (*e.g.*, large email providers). They tend to be well-regarded because they are maintained by reputable providers and incorporated into many email server's. Blacklists are only a static snapshot of spamming hosts, but over time, IP addresses are listed, delisted, and re-listed. It is precisely this history that PreSTA leverages in generating IP reputation.

Filtering based on blacklists alone is imperfect [25]. Listing latency is a commonly cited weakness [20], as is incompleteness. One study reported that 10% of spamming IPs observed at a spam-trap were not blacklisted [23]. Such situations motivate PreSTA; in these partial knowledge scenarios, an unlisted IP address can be viewed in terms of its previous listings (if any) and its spatial relation to other known spamming IPs.

The non-uniform distribution of spamming IPs on the Internet is a well-studied phenomenon. Spamming IPs tend to be found near other spamming IPs [23] and in small regions of the address space [21]. Most such IPs tend to be short-lived [28]; further supporting the use of spatial relationships. Although PreSTA employs basic spatial measures in its preliminary implementation, more advanced relationships could be exploited, such as those suggested in [11, 19]. Additionally, dynamically shaped groups could be used [27].

A key difference between PreSTA and similar work is its combination of temporal history provided by blacklists and the spatial dynamics of spamming IPs. Perhaps the closest related system is SNARE by Hao *et al.* [11]. In addition to demonstrating interesting spatial measures (including geographic distance), SNARE utilizes *simple* temporal metrics to perform spam filtering (*e.g.*, the time-of-day an email was sent) and applies a lightweight form of aggregation (*e.g.*, mean and variance) to detect abnormal patterns. In contrast, PreSTA's temporal computation has more depth, aggregating time-decayed compounding evidence that encodes *months* of *detailed* blacklisting events. Indeed, [11] identifies many valid measures of spamming behavior, but is incapable of Internet-wide scalability due to a reliance on high-dimensional learning. PreSTA spam detection computes over a single feature, IP address (and groups thereof), and is extremely scalable with high accuracy.

Similar techniques are claimed by two commercial services: Symantec [26] uses "IP reputation" in its security software, and SenderBase [12] by Ironport uses spatial data to build IP reputations. The procedures are proprietary, so a detailed comparison is not possible. However, the binary output of the public-facing query mechanisms correlate well with PreSTA's classifications.

PreSTA can also be examined in the context of general-purpose reputation systems/logics, such as EigenTrust [16] or TNA-SL [14]. A key difference involves the nature of feedback; namely, PreSTA considers only negative feedback. Conventional algorithms aggregate over both positive and negative feedback, and feedback is indefinitely retained and associated with a single *discrete* event. PreSTA utilizes *expiring feedback*, where a negative observation (*e.g.*, sending spam) is valid for some finite duration (the blacklist period), after which, it is discarded.

3. REPUTATION MODEL

Although our presentation of PreSTA is focused on the domain of spam detection, it is important to note that PreSTA defines a general reputation model. There are two requirements for potential applications: (1) Access to a history of negative feedback (as achieved via IP blacklists); and (2) the ability to define spatial partitions over entities (as achieved via the IP address hierarchy). The reputation values computed consider both the history of negative feedback for an individual entity and those of related entities.

In the temporal dimension, a history of negative feedback, stored in a *feedback database*, is required. An entity is considered *active* in the database when an associated negative feedback has been recently received (*i.e.*, the entity is listed on the blacklist). After some interval, the feedback expires, and the entity is considered *inactive* (*i.e.*, the entity is delisted from the blacklist). A query to the database returns an entire history of active and inactive events, to which a decay function is applied. The function weighs distant and recent events appropriately and permits compounding evidence to accumulate against entities.

A set of *grouping functions* define spatial relevance. A grouping function maps an entity to other entities that share behavioral properties. More than one grouping function can (and should) be defined, and they may be singular in nature (*i.e.*, an entity is in a group by itself). The temporal history of each spatial grouping is considered, resulting in multiple reputation values. These component reputations are then combined so that a single entity is evaluated based on multiple contexts of negative feedback.

In the remainder of this section the model is formalized. First, the computation and its normalization are discussed, and following that, the feedback database is presented.

3.1 Reputation Computation

The goal of the reputation computation is to produce a quantified value that captures both the spatial and temporal properties of the entity being evaluated. Spatially, the size of the grouping must be considered, and temporally, the history of negative feedback must be weighted in proportion to its spatial relevance.

To capture these properties, three functions are required – two temporal and one spatial:

- $\boldsymbol{hist(\alpha, G, H)}$ is a temporal function returning a list of pairs, (t_{in}, t_{out}), representing listings from the feedback history, H, according to the grouping of entity α by grouping function G. The values t_{in} and t_{out} are time-stamps bounding the active duration of the listing. Active listings return $(t_{in}, \perp)$.

- $\boldsymbol{decay(t_{out}, h)}$ is a temporal function that exponentially decays input times using a half-life h, and it takes the form $2^{-\Delta t/h}$ where $\Delta t = t_{now} - t_{out}$ is of the same unit as h. It returns a value in the range $[0, 1]$, and for consistency, $decay(\perp, h) = 1$.

- $\boldsymbol{size(\alpha, G, t)}$ is a spatial function returning the magnitude, at time t, of the grouping defined by G, of which α is/was a member. If G defines multiple groupings for α, only the magnitude of one grouping is returned. The choice of group is application specific.

Raw reputation can be defined as follows:

$$\boldsymbol{raw\_rep(\alpha, G, H)} = \sum_{\substack{(t_{in},t_{out})\in \\ hist(\alpha,G,H)}} \frac{decay(t_{out}, h)}{size(\alpha, G, t_{in})} \quad (1)$$

This computation captures precisely the spatio-temporal properties required by PreSTA. Temporally, the listing history of an entity/group is captured at each summation via the *hist*() function, and events occurring recently are more strongly weighted via the *decay*() function. Spatially, grouping function G defines the group membership, and each summation is normalized by the group size.

When two or more grouping functions are defined over the entities, multiple computations of *raw_rep*() are performed. Each value encodes the reputation of an entity when considered in a different spatial context. How to best combine reputation is application specific, and for the spam application, machine learning techniques are used (see Sec. 5.7).

The values returned by *raw_rep*() are strictly comparable for all spatial groupings defined by G and the history H. High values correspond to less reputable entities and vice-versa. However, it is more typical for reputation systems [14, 16] to normalize values onto the interval $[0, 1]$ where lower values correspond to low reputation and vice-versa. Ultimately, machine learning does not require normalized values. Such values do, however, enable the model to be consistent with other reputation systems and provide an absolute interpretation that permits manually-authored policies (*e.g.*, allow access where $reputation > 0.8$).

Normalization requires knowledge of an upper bound on the values returned by *raw_rep*(). This cannot be generally defined when the de-listing policy is non-regular. However, if listings expire after a fixed duration d (or a greatest lower-bound for d can be computed), then it is possible to compute an upper bound. Such a bound is found by considering an entity who is as bad as possible; one that is re-listed immediately after every de-listing, and thus, is always active in the feedback database. Considering a grouping of size 1, the *raw_rep*() computation reduces to a geometric sequence:

$$\texttt{MAX\_REP} = 1 + \frac{1}{1 - 2^{-d/h}} \quad (2)$$

Similarly, the same worst case reputation occurs for groups of larger size, however, instead of a single entity acting as a bad as possible, the entire group is simultaneously re-listed immediately following each de-listing. Normalized reputation is now defined as:

$$\boldsymbol{rep(\alpha, G, H)} = 1 - \left(\frac{raw\_rep(\alpha, G, H)}{\texttt{MAX\_REP}}\right) \quad (3)$$

This reputation computation can be modified depending on the entities being evaluated or the nature of the negative feedback database. For example, one can eliminate spatial relevance by using grouping functions that define groups of size 1. Or, one can eliminate all temporal aspects by defining the return of *decay*() as a constant (C). Both such usages are later employed in spam detection; the former due to dynamism in IP address assignment, and the latter due to properties of the blacklist in question. Note that when $decay(t_{out}, h) = C$, $\texttt{MAX\_REP} = decay(\perp, h) + C$.

3.2 Feedback Database

The feedback database, H, depends on the nature of feedback available. PreSTA is most adept at handling *expiring* feedback like that present in IP blacklists. By definition, an expiring feedback occurs when an entity is active (listed) in the database before removal (de-listed) after a finite duration. In this case, H is a record of the entries/exits of listings such that the active database can be reproduced at any point in time.

Feedback can also be *discrete*, where negative feedbacks are associated with a single time-stamp. This is the model most often seen in general-purpose reputation management systems [14, 16]. In such cases, *hist*() always returns pairs of the form $(t_{in}, \perp)$, and thus the associated listings do not decay. A discrete database can be transformed into a compatible H by setting an artificial timeout x, (*e.g.,* $(t_{in}, t_{in} + x)$). Further, listings should not *overlap* (*i.e.,* an entity having multiple active listings). Spam blacklists are inherently non-overlapping, and pre-processing can be applied over feedbacks when this is not the case.

4. SPAM DETECTION SETUP

As presented, PreSTA defines a general model for reputation. Here, we apply PreSTA for the purpose of spam detection. Two properties of spam and IP blacklists are well leveraged by PreSTA. First, spammers are generally found "near" other spammers, and their identifiers, IP addresses, can be spatially grouped based on the IP address hierarchy. Second, blacklists are a rich source of temporal data.

It should be noted that other sources of negative feedback besides IP blacklists could be employed by PreSTA. Any manner of negative feedback associating spamming and IP addresses is sufficient. IP blacklists, however, are a well-regarded and generally trusted source of negative feedback. They are centrally maintained and reputation computed over them can be seen as a good global quantifier. IP blacklists do have weaknesses, and readers should take care not to associate these flaws to the PreSTA model.

4.1 Data Sources

***Blacklists*:** To collect blacklist data, we subscribed to a popular blacklist-provider, Spamhaus [7]. The arrival and exit of IP addresses listed on three Spamhaus blacklists (updated at thirty-minute intervals) were recorded for the duration of the experiment:

- Policy Block List (PBL): Listing of dynamic IP addresses (*e.g.*, those provided by large ISPs such as Comcast or Verizon).
- Spamhaus Block List (SBL): Manually-maintained listing of IPs of known spammers/organizations. Typically these are IPs mapping to dedicated spam servers.
- Exploits Block List (XBL): Automated listing of IPs caught spamming; usually open proxies or machines that have been compromised by a botnet.

As the latter two blacklists contain IP addresses known to have participated in spamming, only these are used to build

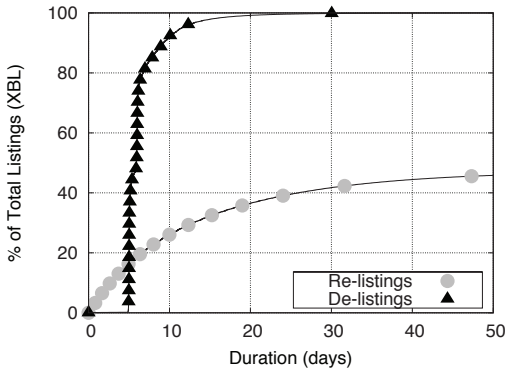


Figure 1: XBL Durations & Re-listing Rates

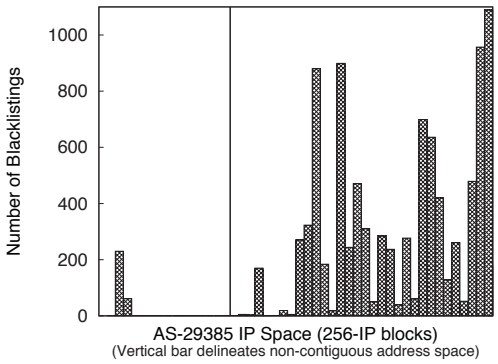


Figure 2: Behavioral Variance within an AS

reputation. The PBL is a preventative measure (however, it is used when examining blacklist performance) which lists hosts that should never be sending email, on principle.

The mechanism by which a blacklist entry occurs, be it accurate or otherwise, is beyond the scope of this work. Removal from the blacklist takes two forms: manual de-listing and timed-expiration. Given its rigorous human maintenance, the SBL follows the former format. The XBL, on the other hand, defaults to a more automated time-to-live de-listing policy. Empirical evidence shows the bulk of such listings expire 5-days after their appearance (see Fig. 1). However, in the case a blacklisted party can demonstrate its innocence or show the spam-generating exploit has been patched, manual removal is also an option for the XBL. Manual de-listings can complicate the calculation of `MAX_REP`, but as we will show, worst case spamming behaviors are rarely realized, permitting strong normalization.

***AS Mappings*:** For the purpose of mapping an IP address to the Autonomous System(s) (AS(es)) that *homes* or *originates* it, CAIDA [2] reports are used. These are compiled from Route Views [8] data and are essentially a snapshot of the BGP routing table.

***Email Set*:** The timestamp and connecting IP address of approximately 31 million email headers were collected at the University of Pennsylvania's engineering email servers between 8/1/2009 and 12/31/2009. The servers host approximately 6,100 accounts, of which roughly 5,500 serve human-users, while the remaining are for various administrative and school uses (*e.g.*, aliases, lists, *etc.*).

A considerable number of emails (2.8 million) in the dataset were both sent and received within the university network. Such emails are not considered in the analysis. Many intra-network messages are the result of list-serves/aliasing, and by excluding them, only externally arriving emails are considered. Our working set is further reduced to 6.1 million emails when analysis is conducted "above the blacklist," or those mails not currently listed on a blacklist (see Sec. 5.1).

A Proofpoint [6] score was provided with each email to categorize it as either spam or ham (not spam). Proofpoint is a commercial spam detection service employed by the University whose detection methods are known to include proprietary filtering and Bayesian content analysis [24] similar to that employed by SpamAssassin [1]. Proofpoint claims extremely high accuracy with a low false-positive rate. Given no other consistent scoring metric and a lack of access to the original email bodies, the Proofpoint score is considered the ground truth in forthcoming analysis.

4.2 Temporal Properties of Spamming IPs

PreSTA leverages the temporal properties of IP blacklists by aggregating the de-listings and re-listings of blacklist entries. Fig. 1 displays the analysis of those two statistics. Of IP-addresses de-listed during the experiment period, 26% were re-listed within 10 days. Overall, 47% of such IPs were re-listed within 10 weeks, and it is precisely such statistics that motivate PreSTA's use of temporal data.

Given that IP addresses are frequently re-listed, we examined the rate at which de-listing occurs; 80% of XBL entries were de-listed at, or very close to, 5 days after their entry (Fig. 1). Even so, this 5-day interval is not fixed. Despite a non-exact expiration, `MAX_REP` is well computed using $d = 5$ (days). Raw reputation values rarely exceeded the calculated `MAX_REP` (less than 0.01% of the time).

The SBL requires a manual confirmation of innocence before de-listing can occur and has no consistent listing length. Thus, `MAX_REP` computation cannot proceed as with the XBL. Instead, the strong assurance provided by de-listing events can be leveraged in reputation calculation. A de-listed IP was verified to be non-spamming, and so there is no reason to decay entries as they exit the list. Formally, $\forall t_{out},\ decay(t_{out}) = 0$, but as previously, $decay(\perp) = 1$. In such circumstances, the `MAX_REP` value for such IPs is computed as 1 (*i.e.*, the IP address is currently listed).

Adjusting the $decay()$ function in this way permits the reputations' of SBL IPs to be based solely on spatial properties. This is a feature of the reputation model, as it allows for flexibility in weighing context when it comes to spatial and temporal information. In a similar way, one can focus solely on temporal properties by defining singular groups, and both produce useful spam classifications (see Sec. 5.7).

4.3 Spatial Properties of Spamming IPs

The hierarchical nature of IP address assignment provides natural spatial groupings for use by PreSTA. Starting at the lowest level, a local router or DHCP service assigns IP addresses to individual machines. The selection pool is likely well-bounded to a subnet (*i.e.,* a /24 or /16). In turn, these routers operate within an ISP/AS, which get their allocations from Regional Internet Registries (RIRs), whose space is delegated from the Internet Assigned Number Authority [4] (IANA). A clear hierarchy exists, and at each level, a

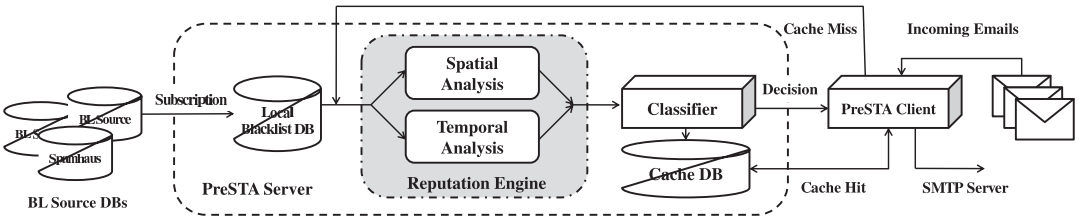


Figure 3: PreSTA **Spam Detection Architecture**

unique reputation can be applied. We focus our groupings at the following three levels: (1) the AS(es) that home(s) the IP, (2) the 768-IP block membership (a rough approximation of a subnet), and (3) the IP address itself.

Despite its easily partitioned nature, it remains to be shown that the IP assignment hierarchy provides relevant groupings. Previous work and anecdotal evidence suggest that AS-number is one of the strongest identifiers of spammers. Indeed, entire AS/ISPs, such as McColo [17] and 3FN [18], have been shut down as a result of their malicious nature. Moreover, in [11], AS-level identifiers were used as a reliable indicator of spamming hosts – indicating that 20 ASes host nearly 42% of spamming IPs.

At the subnet level, it was found that groupings of 768 IP-addresses (*i.e.*, three adjacent /24s) well contain malicious activity (see Sec. 5.5 for details). Fig. 2 visualizes the quantity of of XBL listings in /24 blocks of the address space for an ISP in Uzbekistan. Clearly, there is strong variance across the address space – some regions are highly listed while others are not. The AS-level reputation of this ISP is comparatively poor due to the quantity of listings, but within the address space, certain block-level reputations are ideal. This suggests that AS-level reputation alone may be too broad a metric.

Finally, using a grouping function that singularly groups entities effectively removes spatial relevance from reputation computation. Intuitively, the reputation of a single IP address should be considered because many mail servers use static addresses. However, the often dynamic nature of address assignment implies that unique IP addresses are not singular groupings, but rather, could represent many different machines over time. A recent study reported that the percentage of dynamically assigned IP addresses[1] on the Internet is substantial and that 96% of mail servers using dynamic IPs send spam almost exclusively [31].

5. SPAM IMPLEMENTATION

In this section the implementation of PreSTA for spam detection is described. It is designed with three primary goals: It should produce a classifier that is (1) lightweight; (2) capable of detecting a large quantity of spam; and (3) do so with a low false-positive rate. Design decisions are justified with respect to these goals. Further, the practical concerns of such an implementation are discussed.

The work-flow begins when an email is received and the connecting IP address and timestamp are recorded. Assuming the IP is not actively blacklisted, PreSTA is brought to bear. The IP is mapped to its respective spatial groupings: itself, its subnet, and its originating AS(es). Reputations are calculated at each granularity and these component reputations are supplied as input to a machine-learning classifier trained over previous email. The output is a binary ham/spam label along with each of the three component reputations – all of which may be used by a client application. This procedure is now described in detail, and a visual reference of the PreSTA work-flow is presented in Fig. 3.

5.1 Traditional Blacklists

In Sec. 4.1 the Spamhaus blacklists were introduced. They not only provide the basis on which reputations are built, but in an implementation of PreSTA, it is natural to apply them as intended – to label emails originating from *currently active* IPs as spam. When applied to the email data-set, the blacklists (PBL included) captured 91.0% of spam with a 0.74% false-positive rate. This detection rate is somewhat higher than previous published statistics[2] [15].

Had the intra-network emails not been excluded from analysis, the blacklists would have captured a similar 90.9% of spam emails with a much-reduced 0.46% false-positive rate. The exclusion of such emails, while inflating false-positive rates, permits concentration only on the more interesting set of externally-received emails and does not bias results. The usage of blacklists (independent of spatio-temporal properties), enables fast detection of a large portion of spam emails with minimal time and space requirements – the active listing requires roughly 100MB of storage.

Given the temporal statistics presented in Sec. 4.2, we also experimented with increasing the blacklists' listing period to determine if simple policy changes could greatly affect blacklist performance. This was not the case; increasing the active duration of expired listings (but not those suspected of being manually de-listed) by 5 days increased the detection rate less than 0.05%, and longer listing durations show minimal accuracy improvements at the expense of significant increases in false-positive rates.

5.2 Historical Database

Before reputation can be calculated, a historical feedback database must be in place. As described, Spamhaus blacklists are retrieved at 30-minute intervals. The `diff` is calculated between consecutive copies and time-stamped entries/exits are written to a database. When a new listing appears, the spatial groups (IP, subnet, and AS(es)) that IP is a member of are *permanently* recorded. For example, if IP i was blacklisted as a member of AS a, that entry will always be a part of a's blacklist history.

Roughly 1GB of space is sufficient to store one month's blacklist history (the XBL has 1.0–1.5 million IPs turn over on a daily basis). Fortunately, an extensive history is not

[1]Recall that Spamhaus' PBL blacklist is essentially a listing of dynamic IP addresses. It is constructed mainly using ISP-provided data, and as such, is far from a complete listing.

[2]Our analysis of blacklist performance is from a single-perspective and may not speak to global effectiveness.

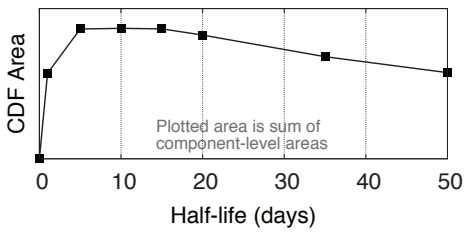


Figure 4: Affect of Half-Life on CDF Area

required given the exponential *decay*() function[3]. For example, given a 10-day half-life, a 3-month old XBL entry contributes 0.6% the weight of an active listing. Lengthy histories offer diminishing returns. To save space, one should discard records incapable of contributing statistical significance. Further, such removal saves computation time because the smaller the set *hist*() returns, the fewer values which must be processed by *raw_rep*().

5.3 Grouping Functions

Given an entity (IP address) for which to calculate reputation, three grouping functions are applied:

- IP Function: An IP is a group in and of itself, so such a grouping function mirrors its input.
- Subnet Function: IP subnet boundaries are not publicly available. Instead, an estimate considers blocks of IP addresses (we use the terms "subnet-level" and "block-level" interchangeably). IP space is partitioned into /24s (256 IP segments), and an IP's block grouping consists of the segment in which it resides as well as the segment on either side; 768 addresses per block. Thus, block groupings overlap in the address space, and a single IP input returns one block of IPs (three /24s). Although such estimations may overflow known AS boundaries, these naïve blocks prove effective.
- AS Function: Mapping an IP to its parent AS(es) requires CAIDA [2] and RouteViews [8] data. Note that some AS boundaries overlap in address space and some portions of that space (*i.e.,* unallocated portions) have no resident AS whatsoever. An IP can be homed by any number of ASes, including none at all, the technical considerations of which are addressed in Sec. 5.5. The function's output is all the IPs homed by an AS(es) in which the input IP is a member. Each returned IP is tagged with the parent AS(es), so a well-defined subset of the output can be chosen.

5.4 Decay Function

The decay function (Sec. 3.1) controls the extent to which temporal proximity factors into reputation. It is configured via its half-life, h. If h is too small, reputations will decay rapidly and provide little benefit over using blacklists alone. Too large an h will cause an increase in false positives due to stale information.

[3]This minimal history requirement was of benefit to this study. Reputations must *warm-up* before their use is appropriate. Indeed, collection of blacklist data began in 5/2009, three months before the first classifications.

A good half-life will maximize the difference between the reputations of spam and ham email. Analyzing email predating the evaluation period, the reputation-CDFs for both spam and ham emails (as in Fig. 6) were plotted using different h, seeking to maximize the area between the curves. In Fig. 4 the calculations from these experiments are presented. A value of $h = 10$ (days) was found optimal and this value is used in the spam application[4]. With the half-life established and having chosen $d = 5$ (days), `MAX_REP`$= 4.14$.

As described previously, two separate *decay*() functions are employed depending on whether a listing appeared on the SBL or the XBL. Manually maintained, de-listing from the SBL is not decayed, but the XBL is decayed using the aforementioned 10-day half-life. A special flag attached to each time pair returned by *hist*() allows both listings to be used in combination.

5.5 Reputation Calculation

Given the feedback database (Sec. 5.2), output (sets of IP addresses) of the three grouping functions (Sec. 5.3), and the decay function (Sec. 5.4), reputation may now be calculated at each granularity, returning three reputation values. Calculation closely follows as described in Sec. 3.1.

Calculation of IP-level and subnet-level reputation is straightforward per the reputation model with $size() = 1$ and $size() = 768$, respectively. The particulars of AS-level calculation are more interesting. An IP may be a member of any quantity of ASes, including none at all. If an IP is multi-homed, the conservative choice is made by selecting the most reputable AS-level reputation. Those IPs mapping to no AS form their own group, and the reputation for this group is designated as 0 because, in general, unallocated space is only used for malicious activity (see Sec. 7). In this spatial grouping, *size*() is not constant over time. Instead, magnitudes are pre-computed for all AS using CAIDA data and updated as BGP routes change.

5.6 Calculation Optimizations

PreSTA must calculate reputation efficiently to achieve the desired scalability. It should not significantly slow email delivery (latency), and it should be capable of handling heavy email loads (bandwidth). Caching strategies and other techniques that support these goals are described below:

- AS Value Caching: Reputations for *all* ASes are periodically recalculated off-line. Calculation is (relatively) slow given that *hist*() calls return large sets.
- Block/IP Value Caching: Similarly, block and IP reputations can be cached after the first cache miss. Cache hit rates are expected to be high because (1) an email with multiple recipients (*i.e.,* a carbon copy) is received multiple times but with the same source IP address, and (2) source IP addresses are non-uniformly distributed. For the 6.1 million (non-intra-network, non-blacklisted) emails in the working data-set, there are 364k unique IP senders and 176k unique 'blocks.'
- Cache Consistency: Caches at all levels need to be flushed when the blacklists are updated (every 30 minutes), to avoid inconsistencies involving the arrival of

[4]Although it was found unnecessary, h could be optimized on an interval basis, much like re-training a classifier. However, experiments showed minor variations of the parameter to be inconsequential.

new listings. As far as time-decay is concerned, a discrepancy of up to 30 minutes is inconsequential when considering a 10-day half-life.

- WHITELISTING: There is no reason to calculate reputation in trusted IP addresses, such as one's own server. Of course, whitelists could also be utilized in a feedback loop to alleviate false-positives stemming from those entities whose emails are misclassified.

Using these optimizations, the PRESTA implementation is capable of scoring 500k emails an hour, with average email latency on the order of milliseconds[5]. Latency and bandwidth are minimal concerns. Instead, it is the off-line processing supporting this scoring which is the biggest resource consumer. Even so, the implementation is comfortably handled by a commodity machine and could easily run adjacent to an email server. Pertinent implementation statistics, such as cache performance, are available in Sec. 6.4.

5.7 Reputation Classification

Extraction of a binary classification (*i.e.,* spam or ham) is based on a *threshold* strategy. Emails evaluated above the threshold are considered ham, and those below are considered spam. Finding an appropriate threshold can be difficult, especially as dimensionality grows, as is the case when classifying multiple reputation values. Further, a fixed threshold is insufficient due to temporal fluctuations; as large groups (botnets) of spamming IPs arise and fall over time, the distinction between good and bad may shift.

A *support vector machine* (SVM) [13] is employed to determine thresholds. SVM is a form of supervised learning that provides a simple and effective means to classify multiple reputation values. The algorithm maps reputation triples (a feature for each spatial dimension) from an email training set into 3-dimensional space. It then determines the surface (threshold) that best divides spam and ham datapoints based on the training labels. This same threshold is then applied during classification. The SVM routine is tuned via a *cost* metric that is correlated to the eventual false-positive rate of the classifier.

The classifier is adjusted (re-trained) every 4 days to handle dynamism. A subset of emails received in the previous 4 days are trained upon, and the resulting classifier is used for the next 4 day interval. The affect of different training periods has not been extensively studied. Clearly, large periods are not desired; the reputation of distant emails may not speak to the classification of current ones. Too short a period is poor because it requires extensive resources to retrain so frequently. Analysis found 4-day re-training to be a good compromise. However, the re-training period need not be fixed, and future work will explore re-training rates that adjust based on various environmental factors.

At each re-training, 10,000 emails (5% of the non-intra-network, non-blacklisted email received every 4 days) were used, and emails were labeled as spam/ham based on the Proofpoint score. In a more general use case, there would be some form of client feedback correlated across many accounts that can classify spam post-delivery and train various spam detectors. Since we do not have access to such user behavior, correlation statistics, or any external spam filters, the provided Proofpoint values are assumed.

[5]Statistics are based on a single-threaded implementation. Concurrency and other programming optimizations would likely improve PRESTA's performance and scalability.

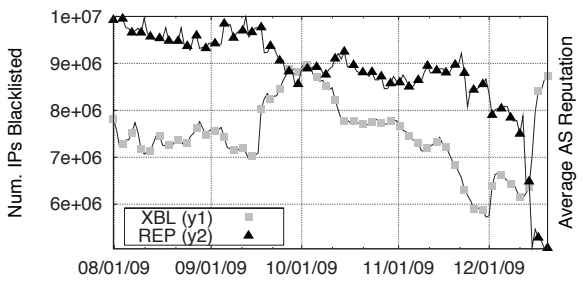


Figure 5: XBL Size Relative to Global Rep.

Post-training, the false-positive (FP) rate of the classifier is estimated by measuring the error over the training set (assuming one does not over-fit the training data). The estimated FP-rate is a good indicator of the true FP-rate, and the SVM cost parameter is adjusted to tune the expected FP-rate. All classifier statistics and graphs hereafter were produced with a 0.5% tolerance for false-positives (over the classification set), as this simplifies presentation. This FP-rate (0.5%) is a reasonable setting given that blacklists are widely accepted and achieved a 0.74% FP-rate over the same dataset. Additionally, these rates are somewhat inflated given the decision to exclude intra-network emails, which are unlikely to contribute false-positives (the blacklist FP-rate was reduced one-third to 0.46% with their inclusion). In Sec. 6.5, the trade-off between the FP-rate and spam blockage is examined in greater depth.

6. EXPERIMENTAL ANALYSIS

Experimental analysis begins by examining component reputations individually. From there, two case studies are presented which exemplify how PRESTA produces metrics outperforming traditional blacklists in both spatial and temporal dimensions. Finally, the detection results of the PRESTA spam filter are presented.

To best simulate a real email server load, it is assumed emails arrive in the order of their timestamps and are evaluated relative to this ordering. Additionally, cache population/flushing and classification re-training are performed at the relative time-intervals outlined in the previous section.

6.1 Blacklist Relationship

In examining how reputations quantify behavior, we apply a simple intuition: One would expect to see a clear push-pull relationship between an entity's reputation and the number of corresponding entries on the blacklist. To confirm this hypothesis, the size of the XBL blacklist[6] was graphed over time and compared to the average reputation of *all* ASes. Results are presented in Fig. 5. An inverse relationship is observed, confirming the hypothesis. When the number of listings decreases, reputation increases – and vice versa.

6.2 Component Reputation Analysis

In order for component reputations (IP, block, and AS) to be useful in spam detection they must be *behavior predictive*. That is, the reputations of ham emails should exceed those

[6]The XBL is the driving force behind reputation. The SBL is also a contributor, but is orders of magnitude smaller.

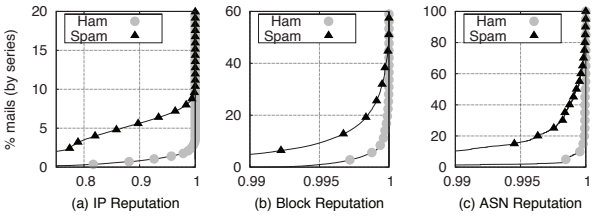


Figure 6: CDFs of Component Reputations

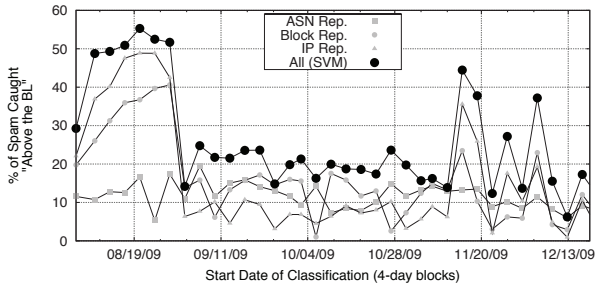


Figure 7: Component Reputation Performance

of spam emails. This relationship is visualized in the CDFs of Fig. 6. All component reputations behave as expected. Fig. 6 also displays the benefit of multiple spatial groupings. While 90% of spam emails come from IPs that had ideal reputation (*i.e.,* a reputation of 1) at the time of receipt, this is true for just 46% of blocks, and only 3% of AS.

The CDFs of Fig. 6 imply that each component reputation is, in and of itself, a metric capable of classifying *some* quantity of spam. However, it is desirable to show that each granularity captures *unique* spam, so that the combination of multiple reputations will produce a higher-order classifier of greater accuracy. In Fig. 7, the effectiveness of each component reputation is presented. The percentage of spam caught is "above the blacklist," or more precisely, the percentage of spam well-classified by the reputation value that was not identified by the blacklist alone[7]. Crucially, the combined performance (the top line of Fig. 7), exceeds that of any component, so each spatial grouping catches spam the others do not. On the average, PreSTA is able to capture 25.7% of spam emails not caught by traditional blacklists.

We are also interested in determining which grouping provides the best classification. AS-level reputation is the most stable of the components, individually capable of classifying an additional 10-15% of spam above the blacklist. However, during periods of increased PreSTA performance, it is often the block and IP levels that make significant contributions. This is intuitive; AS-level thresholding must be conservative. Given their large size, the mis-classification of an AS could result in an unacceptable increase in the FP-rate. Meanwhile, the cost associated with a mis-prediction is far less for block and IP groupings.

These results suggest that considering more spatial dimensions should increase performance, that is, when there are non-overlapping classifications. However, there are diminishing returns. Each additional component reputation requires increased resources in evaluation and classification.

[7]Given the inclusion of traditional blacklist filtering, the primary concern is those emails that are not actively listed.

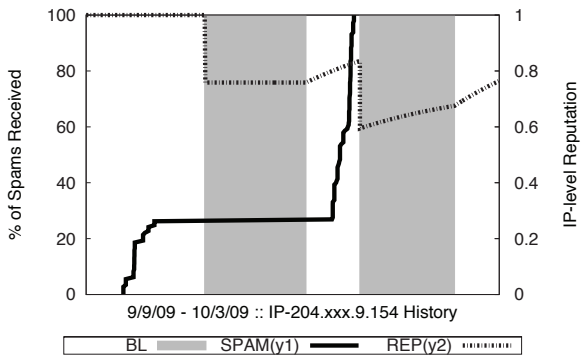


Figure 8: Single IP Behavior w.r.t. Blacklisting

An application should seek a minimal set of dimensions to best represent and classify its data.

6.3 Case Studies

Two case studies exemplify the types of spam behavior able to evade blacklists, yet captured via PreSTA. First, Fig. 8 shows the *temporal* sending patterns of a single spamming IP address. This IP was blacklisted twice during the course of the study (as indicated by shaded regions), and the time between listings was small ($\approx$ 2 days). The controller of this IP address likely used blacklist counter-intelligence [22] to increase the likelihood that spam would be delivered: Notice that no spam was observed when the IP was actively listed, but 150 spam emails were received at other times.

Traditional blacklist are reactive, binary measures that do not take history into account. During the intermittent period between listings, as far as the blacklist is concerned, the spamming IP is an innocent one. However, as shown in Fig. 8, the IP-level reputation metric compounds prior evidence. Thus, if PreSTA had been in use, the intermittent influx of email likely would have been identified as spam.

Secondly, Fig. 9 visualizes a case study at the AS-level utilizing both *spatial* and *temporal* dimensions. In the early stages of data collection anomalous activity was noticed at a particular AS (AS#12743)[8]. Even when compared to the other four worst performing ASes during the time block, ASN-12743's drop in reputation is astounding. Nearly its entire address space, some 4,500 addresses, were blacklisted over the course of several days – likely indicative of an aggressive botnet-based spam campaign. After this, the reputation increases exponentially (per the half-life), eventually returning to innocent levels.

With traditional blacklists, an IP must actually send spam before it can be blacklisted. In the ASN-12743 case, this means all 4,500 IPs had some window in which to freely send spam. However, as the IPs were listed in mass, the *reputation* of the AS drops at an alarming rate, losing more than 50% of its value. Had PreSTA been implemented, the reputation of the AS (and the blocks within) would have been low enough to classify mails sourced from the remainder of the space as spam, mitigating the brunt of the attack.

6.4 Implementation Performance

An important aspect of PreSTA is its scalability, and to best evaluate this our PreSTA simulation mimicked the

[8]PTK-Centertel, a major Polish mobile service provider.

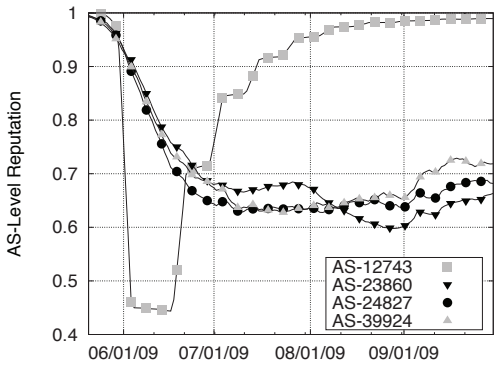


Figure 9: Temporal Shift within Grouping (AS)

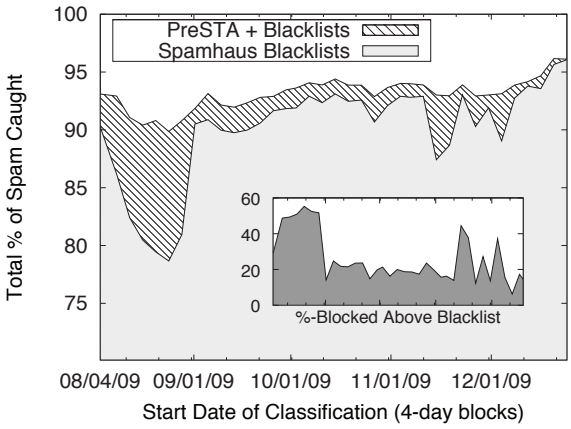


Figure 10: Blacklist and PreSTA Blockage

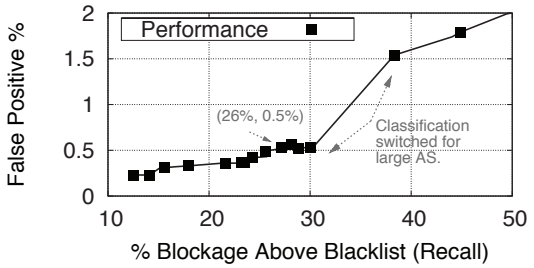


Figure 11: Characteristic ROC Trade-Off

normal processing of a mail server. The blacklist history and cached reputation scores were regulated so that only the knowledge available at the time of arrival is used to evaluate an email. PreSTA requires a warm-up period to gather enough temporal knowledge to process correctly; hence, historical blacklist storage began three months prior to the first email being scored.

The effectiveness of the cache and the latency of the system is also of interest. Caching is highly effective: 56.8% of block-level calculations are avoided, and 43.1% are avoided at the IP-level (recall that *all* AS-level calculations are performed off-line and then cached). The reputation of an incoming email is calculated in nearly real time, with the average email being processed in fractions of a second. Under typical conditions, over 500,000 emails can be scored in an hour, using commodity hardware.

Re-training the classifiers and rebuilding the AS-cache are the most time consumptive activities. Fortunately, training new classifiers takes only minutes of work for a 10,000 email training set, and only needs to be performed every 4 days. Re-training is also done off-line and does not affect current scoring. Rebuilding the AS reputation cache must be done every 30 minutes, once new blacklist data is available, but it need not delay current scoring as the previous AS-level reputations are still relevant (at most 30 minutes old).

6.5 Spam Mitigation Performance

The spam detection capabilities of PreSTA are summarized in Fig. 10. On average, 93% of spam emails are identified when used in conjunction with traditional blacklists. This may seem to be a nominal increase over blacklists alone; however, the inset of Fig. 10 is more intuitive – PreSTA blocks between 20% and 50% of those mails passing the Spamhaus blacklists, with a 25.7% average (identical to the top line of Fig. 7). Had PreSTA been implemented on our university mail server during the study, it would have caught 650,000 spam emails that evaded the Spamhaus blacklists.

Most interestingly, PreSTA provides consistent and steady state detection. For example, consider the significant variations in blacklist performance seen throughout the study (for example, in late August 2009 and in mid-November 2009). PreSTA is nearly unaffected during these periods and may be a useful stop-gap during variations in blacklist accuracy. While the blockage-rates of the blacklists fluctuate 18% over the course of the study, PreSTA is far more consistent, exhibiting just 5% of variance. Further, it is likely that continued analysis will show similar variations in blacklist performance. Periods of high de-listing are likely followed by periods of high re-listing as spammers try to maximize the utility of available IPs.

Ultimately, the performance attainable by the classifier is dependent on the number of false-positives (FPs) tolerated. To this point, the FP-rate has been fixed at 0.5%; however, as exemplified in Fig. 11, the FP-rate is tune-able and strongly correlates with the blockage rate. The plot is generated over a characteristic interval of email from mid-October 2009, and is akin to the precision/recall graphs common in machine-learning. We remind readers that the decision to exclude intra-network emails from the dataset (see Sec. 5.1) significantly inflates the presented FP-rates.

7. EVASION AND GAMESMANSHIP

To be effective, PreSTA must be robust to evasion and gamesmanship – an entity should be unable to surreptitiously influence its own reputation. Given the use of IP blacklists as a feedback source, the most effective way to avoid PreSTA is to avoid getting blacklisted. However, such a technique is not fail-safe; a single evasive entity may still have poor reputation at broader granularity. When negative feedback exists, and an IP has been blacklisted, the best recourse is patience. Over time, the weight of the listing decays. As such, there is no way to evade PreSTA in the temporal dimension.

However, spammers are migrant and the spatial dimension affords greater opportunities. While IP and block magnitudes are fixed, an AS controls the number of IPs it broadcasts. An actively evasive AS would ensure its entire allo-

cation is broadcasted. More maliciously, a spammer may briefly hijack IP space they were *not* allocated in order to send spam from stolen IPs. Such *spectrum agility* was shown by [21] to be an emergent spamming technique. Fortunately, if the hijacked IP space was not broadcasted (*i.e.,* unallocated), emails from these IPs would map to the special grouping "no AS", whose reputation is zero (per Sec. 5.5). However, if the hijacked space was broadcasted by a reputable AS, evasion may be possible. Fortunately, [21] observes the use of unallocated space is most prevalent.

The previous scenario can be described as a *sizing attack* and is of most concern to PreSTA. The entities being evaluated should not be able to affect the size of their spatial groupings. However, this attack is only effective when the group size is non-singular, and an simple mitigation technique is to always include a grouping function defining singular groups. Further, an implementation should assign persistent identifiers to entities. When identifiers are non-persistent, PreSTA could fall victim to a Sybil attack [10] since an entity could evade negative feedback by simply changing identifiers.

8. CONCLUSIONS

In this paper, we have introduced PreSTA, a novel reputation model designed to combine the rich historical information of blacklists and the spatial relationships of spamming IPs. We have shown PreSTA reputations to be an effective measure for classifying spam, identifying up to 50% of spam not caught by blacklists alone. Our preliminary implementation, which combines PreSTA with blacklists, mitigates 93% of spam on average and is stable – reducing the effects of blacklist fluctuations. Finally, PreSTA proves scalable and is able to efficiently handle production email workloads, processing over 500k emails an hour.

Having demonstrated PreSTA's proficiency in the field of spam detection, one must consider how this capability is best utilized. Although we make no claims it can (or should) replace content-based filtering, PreSTA could be applied as an initial filter or grey-listing mechanism. Alternatively, the system could be used to prioritize the processing of incoming email in high-volume situations. Since it is based on centralized blacklist information, PreSTA could be installed as a parallel service provided by blacklist providers.

Further, PreSTA's applicability is broader than email spam. PreSTA has already proven effective in the detection of Wikipedia vandalism [30] and shows promise in other domains ranging from prioritization of BGP announcements to reputation for web-based service *mash-ups*. Any service that requires dynamic decision making and has access to records of historical feedback is a candidate. Ultimately, PreSTA reputations may be utilized as an effective means of performing dynamic access-control and mitigating malicious behavior, two extremely relevant issues as service paradigms shift to more distributed architectures.

9. ACKNOWLEDGMENTS

The authors would like to thank Wenke Lee and David Dagon, both of Georgia Tech, for their initial guidance on this project. Additional thanks go to Charles 'Chip' Buchholtz of UPenn-CETS, who performed mail dumps and aided us in obtaining permission to process those logs.

References

[1] Apache SpamAssassin. http://spamassassin.apache.org/.

[2] CAIDA. http://www.caida.org/.

[3] DNSBL.info: Blacklists. http://www.dnsbl.info/dnsbl-list.php.

[4] Internet Assigned Numbers Authority. http://www.iana.org/.

[5] MessageLabs Intelligence. http://www.messagelabs.com/.

[6] Proofpoint, Inc. http://www.proofpoint.com/.

[7] Spamhaus Project. http://www.spamhaus.org/.

[8] Univ. of Oregon Route Views. http://www.routeviews.org/.

[9] M. Blaze, S. Kannan, A. D. Keromytis, I. Lee, W. Lee, O. Sokolsky, and J. M. Smith. Dynamic trust management. *IEEE Computer (Special Issue on Trust Mangement)*, 2009.

[10] J. Douceur. The Sybil attack. In *1st IPTPS*, March 2002.

[11] S. Hao, N. A. Syed, N. Feamster, A. G. Gray, and S. Krasser. Detecting spammers with SNARE: Spatio-temporal network-level automated reputation engine. In *USENIX Security Sym.*, 2009.

[12] IronPort Systems Inc. Reputation-based mail flow control. White Paper, 2002. (SenderBase).

[13] T. Joachims. *Advances in Kernel Methods - Support Vector Learning*, chapter Making Large-scale SVM Learning Practical, pages 169–184. MIT Press, Cambridge, MA, 1999.

[14] A. Jøsang, R. Hayward, and S. Pope. Trust network analysis with subjective logic. In *Proceedings of the 29th Australasian Computer Science Conference*, 2006.

[15] J. Jung and E. Sit. An empirical study of spam traffic and the use of DNS black lists. In *Proc. of the 4th ACM SIGCOMM Conference on Internet Measurement*, pages 370–375, 2004.

[16] S. D. Kamvar, M. T. Schlosser, and H. Garcia-molina. The EigenTrust algorithm for reputation management in P2P networks. In *Proc. of the Twelfth WWW Conference*, May 2003.

[17] B. Krebs. Host of Internet spam groups is cut off. http://www.washingtonpost.com/wp-dyn/content/article/2008/11/12/AR2008111200658.html, November 2008. (McColo).

[18] B. Krebs. FTC sues, shuts down N. Calif. web hosting firm. http://voices.washingtonpost.com/securityfix/2009/06/ftc_sues_shuts_down_n_calif_we.html, June 2009. (3FN).

[19] Z. Qian, Z. Mao, Y. Xie, and F. Yu. On network-level clusters for spam detection. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, 2010.

[20] A. Ramachandran, D. Dagon, and N. Feamster. Can DNSBLs keep up with bots? In *Proc. of the 3rd CEAS*, 2006.

[21] A. Ramachandran and N. Feamster. Understanding the network-level behavior of spammers. In *Proc. of SIGCOMM 2006*, 2006.

[22] A. Ramachandran, N. Feamster, and D. Dagon. Revealing botnet membership using DNSBL counter-intelligence. In *USENIX: Steps to Reducing Unwanted Traffic on the Internet*, 2006.

[23] A. Ramachandran, N. Feamster, and S. Vempala. Filtering spam with behavioral blacklisting. In *Proc. of Computer and Communications Security (CCS '07)*, pages 342–351, 2007.

[24] M. Sahami, S. Dumais, D. Heckerman, and E.Horvitz. A Bayesian approach to filtering junk e-mail. In *AAAI-98 Workshop on Learning for Text Categorization*, 1998.

[25] S. Sinha, M. Bailey, and F. Jahanian. Improving spam blacklisting through dynamic thresholding and speculative aggregation. In *Proceedings of the 17th NDSS*, 2010.

[26] Symantec Corportation. IP reputation investigation. http://ipremoval.sms.symantec.com/.

[27] S. Venkataraman, A. Blum, D. Song, S. Sen, and O. Spatscheck. Tracking dynamic sources of malicious activity at internet scale. In *Neural Information Processing Systems '09*, 2009.

[28] S. Venkataraman, S. Sen, O. Spatscheck, P. Haffner, and D. Song. Exploiting network structure for proactive spam mitigation. In *16th USENIX Security Symposium*, pages 149–166, 2007.

[29] A. G. West, A. J. Aviv, J. Chang, V. S. Prabhu, M. Blaze, S. Kannan, I. Lee, J. M. Smith, and O. Sokolsky. QuanTM: A quantitative trust management system. In *EUROSEC 2009*, pages 28–35, March 2009.

[30] A. G. West, S. Kannan, and I. Lee. Detecting Wikipedia vandalism via spatio-temporal analysis of revision metadata. In *EUROSEC '10*, pages 22–28, Paris, France, 2010.

[31] Y. Xie, F. Yu, K. Achan, E. Gillum, M. Goldszmidt, and T. Wobber. How dynamic are IP addresses? In *SIGCOMM '07*, 2007.

Breaking e-Banking CAPTCHAs

Shujun Li
University of Konstanz, Germany

S. Amier Haider Shah
National University of Science and Technology, Pakistan

M. Asad Usman Khan
National University of Science and Technology, Pakistan

Syed Ali Khayam
National University of Science and Technology, Pakistan

Ahmad-Reza Sadeghi
Ruhr-University of Bochum, Germany

Roland Schmitz
Stuttgart Media University, Germany

ABSTRACT

Many financial institutions have deployed CAPTCHAs to protect their services (e.g., e-banking) from automated attacks. In addition to CAPTCHAs for login, CAPTCHAs are also used to prevent malicious manipulation of e-banking transactions by automated Man-in-the-Middle (MitM) attackers. Despite serious financial risks, security of e-banking CAPTCHAs is largely unexplored. In this paper, we report the first comprehensive study on e-banking CAPTCHAs deployed around the world. A new set of image processing and pattern recognition techniques is proposed to break *all* e-banking CAPTCHA schemes that we found over the Internet, including three e-banking CAPTCHA schemes for transaction verification and 41 schemes for login. These broken e-banking CAPTCHA schemes are used by thousands of financial institutions worldwide, which are serving hundreds of millions of e-banking customers. The success rate of our proposed attacks are either equal to or close to 100%. We also discuss possible improvements to these e-banking CAPTCHA schemes and show essential difficulties of designing e-banking CAPTCHAs that are both secure and usable.

Categories and Subject Descriptors

K.6.5 [**Computing Milieux**]: Management of Computing and Information Systems—*Security and Protection*

General Terms

Security, Electronic Commerce, Human Factors

Keywords

CAPTCHA, e-banking, man-in-the-middle attack, malware

1. INTRODUCTION

Due to their ease and ubiquity of use, e-banking systems have experienced worldwide deployments. A 2009 survey of the American Bankers Association reveals that e-banking has been the preferred banking method of bank customers [1]. Security of e-banking systems is a major concern for the financial institutions and their customers. The highly sensitive nature of data processed by e-banking systems necessitates a robust security framework to protect the users' privacy, identity and assets.

Many financial institutions around the world have deployed CAPTCHAs[1] to protect their e-banking systems from automated attacks. In addition to traditional CAPTCHAs for preventing automated login attempts, some financial institutions have also deployed CAPTCHAs for transaction verification. The main goal of these CAPTCHAs is to make automated transaction manipulation by malicious programs (e.g., Trojans) more difficult. These CAPTCHAs are supposed to provide security against Man-in-the-Middle (MitM) attacks, which can manipulate the communication between the user and the e-banking server on the fly. Such attacks are much more difficult to detect than credential stealers (like email-based phishing attacks and keyloggers) because they can circumvent many existing e-banking protection mechanisms including multi-factor authentication schemes. While the number of such attacks remains unknown, large-scale attacks are becoming more and more likely with the rising infection rate and evolving sophistication of malware on desktop PCs and smart phones.

Existing e-banking solutions counter MitM attacks by introducing some means of transaction verification into the system. These solutions can be broadly divided into the following classes: trusted out-of-band channel [2]; hardware for establishing encrypted channel over untrusted network and/or computer [3]; hardware with trusted display/keypad for generating transaction-dependent TANs or signatures [4, 5]; and solutions based on CAPTCHAs (see Sec. 2.2). The main advantage of CAPTCHA-based solutions is that they do not depend on special hardware and therefore the implementation and maintenance costs are very low for both financial institutions and customers.

The main premise of e-banking CAPTCHAs for both login and transaction is that some pattern recognition tasks are extremely difficult for computers (e.g., automated programs like malware) but easy for humans. Based on this premise, e-banking systems protected by CAPTCHAs are considered secure against automated attackers which aim to interpret

[1]CAPTCHA is an acronym for "Completely Automated Public Turing test to tell Computers and Humans Apart".

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

or forge CAPTCHAs. A further challenge of breaking transaction e-banking CAPTCHAs is that the automated attack needs to run in real time to avoid being noticed by users.

A number of prior efforts have been made for analyzing the security of general-purpose CAPTCHAs (which are only for login). However, to the best of our knowledge, the security of e-banking CAPTCHAs has not yet been evaluated thoroughly.

Contribution: This paper presents the first comprehensive study on e-banking CAPTCHAs, and shows that existing e-banking CAPTCHAs do not meet the expected security requirements. More precisely, we report practical attacks on three e-banking CAPTCHA schemes for transaction verification and 41 schemes for login.[2] Our attacks are based on a new set of image processing and pattern recognition tools, including k-means clustering [6], digital image inpainting [7,8], morphological image processing [9], character recognition based on cross-correlation [10] and an image quality assessment (IQA) method called CW-SSIM [11]. As far as we know, it is the first time that image inpainting and IQA algorithms are used to break CAPTCHAs. Our attacks are alarmingly successful: all of the e-banking CAPTCHA schemes are broken with a success rate equal to or close to 100%. Most of our proposed attacks can run in real time.

Our further investigation shows that it is nontrivial to enhance the security of the broken e-banking CAPTCHAs. CAPTCHAs have some essential drawbacks rooted in their design principle that makes it difficult to simultaneously achieve both usability and security. We thus call for cautions in deploying e-banking CAPTCHAs.

Outline: The rest of this paper is organized as follows. In the next section, we give a survey of related work on the cryptanalysis of CAPTCHAs and the use of CAPTCHAs in e-banking systems. In Sec. 3 we introduce the new set of CAPTCHA-breaking tools used in our attacks. Section 4 demonstrates how these tools are used to break a typical e-banking CAPTCHA scheme for transaction verification (used by around 800 German banks). Then, Section 5 reports our attacks on another two CAPTCHA schemes for transaction verification, which are deployed by two major banks in China. Section 6 shows that 41 e-banking login CAPTCHA schemes deployed by many financial institutions all over the world cannot resist automated segmentation attacks. Based on the proposed attacks, in Sec. 7 we outline some possible improvements to the broken e-banking CAPTCHA schemes, and discuss whether CAPTCHAs are at all appropriate for protecting e-banking system. The last section summarizes the salient findings of our work.

2. RELATED WORK

2.1 CAPTCHAs in general

A CAPTCHA [12] scheme is a challenge-response authentication protocol based on a hard AI problem, which can be easily solved by humans but not by machines. Here, the goal differs from traditional user authentication protocols: to accept humans but reject automated programs. CAPTCHAs can be designed in many forms. The most well-known and widely-deployed form is distorted texts shown as CAPTCHA images. The distortions are chosen in a way that automated programs cannot achieve a comparable recognition rate to what humans can. Figure 1 shows some CAPTCHAs of this kind. Similarly, audio CAPTCHAs designed for the blind use distorted audio as the challenge shown to the prover.

Figure 1: Three CAPTCHAs based on distorted texts (left to right): Google, Microsoft, Yahoo!

Another form of CAPTCHA is based on hard AI problems on image understanding. A typical CAPTCHA of this kind is Asirra [13], which challenges the prover to select all cat pictures from 12 pictures of cats and dogs. The idea of image-based CAPTCHAs has also been generalized to be based on animation, video, and 3-D models. Readers are referred to [14] for a recent survey on CAPTCHAs.

The idea of breaking CAPTCHAs has been around for a while. The first public report we know appeared in 2003 [15], in which Mori and Malik proposed recognition based attacks on Gimpy and EZ-Gimpy, two early CAPTCHA schemes based on distorted texts. Later, several other attacks were reported, showing insecurity of more CAPTCHA schemes based on distorted texts [16, 17]. Moreover, Hocevar demonstrated results of his attacks on quite a number of CAPTCHA schemes on his web site [18], which reveals some common pitfalls of weak CAPTCHAs. In [19], Chellapilla et al. reported an interesting finding: once a CAPTCHA image based on distorted texts is well segmented, automated programs can recognize those single characters even better than humans. This implies that making segmentation harder is the main way to enhance security of CAPTCHAs based on distorted texts. In [20], Yan and Ahmad showed that a simple pixel-count based attack can break a number of CAPTCHAs offered at Captchaservice.org and deployed by some other web sites. In [21], Yan and Ahmad proposed a new attack on some distorted texts based CAPTCHAs, which can be used to segment CAPTCHA images into single characters with high accuracy.

Most of the existing attacks are designed for CAPTCHA schemes that use distorted texts. There are also some attacks on other kinds of CAPTCHA schemes. In [22], Golle showed that a machine learning based attack can achieve a success rate of 10.3% for a 12-image challenge of the image-based CAPTCHA scheme Asirra. In [23,24], attacks to some deployed audio CAPTCHA schemes were reported.

There are also attacks exploiting implementation flaws. In [25], Hernandez-Castro and Ribagorda proposed a side-channel attack on a CAPTCHA scheme based on solving mathematical problems. In [26], Hindle et al. showed that reverse engineering can help to design new attacks. Recently, Hernandez-Castro and Ribagorda pointed out some common problems of many CAPTCHA schemes [27].

2.2 CAPTCHAs for e-banking

One of the main applications of CAPTCHAs is to prevent automated online password attacks [28]. Therefore, many financial institutions have deployed CAPTCHAs on the login pages of their e-banking systems to protect their customers from such attacks. In addition to login CAPTCHAs, many financial institutions have also deployed CAPTCHAs

[2]These are *all* the e-banking CAPTCHA schemes that we had found when we submitted the final edition of this paper.

for transaction verification, in order to prevent automated MitM attacks. The CAPTCHA-based transaction verification works as follows. After receiving a transaction request, the server generates a CAPTCHA image by embedding the transaction data, a dynamic TAN (Transaction Authentication Number) and probably some other information, which is sent to the user for confirming the transaction. In case an automated MitM attacker cannot recognize the textual information embedded in the CAPTCHA image, it will be unable to forge a CAPTCHA image. Although the security of transaction CAPTCHAs depend on the same principle of login CAPTCHAs, there are some essential differences between transaction CAPTCHAs and login CAPTCHAs: 1) a transaction CAPTCHA image generally contains much more characters than a login CAPTCHA image; 2) some (often most) characters in a transaction CAPTCHA image are known to the MitM attacker; 3) forging CAPTCHA images can also break transaction CAPTCHAs. While there has been a large body of previous work on breaking login CAPTCHAs, transaction CAPTCHAs are unique for e-banking and security analysis of them remains unexplored.

We did not find a comprehensive report on e-banking CAPTCHAs deployed by the worldwide banking sector, so we manually checked web sites of many financial institutions. We found e-banking CAPTCHA schemes deployed by a large number of financial institutions in different countries such as China, Germany, USA, Australia and Switzerland.

Deployment of e-banking CAPTCHAs in China is so popular that it has become a standard components of almost every e-banking system. We checked 30 major Chinese banks, among which almost all have deployed login CAPTCHAs, and at least two have deployed transaction CAPTCHAs. In Germany, the pattern is a bit different: many banks have deployed login CAPTCHAs and some have also deployed transaction CAPTCHAs. A similar pattern is observed for the banking industry in the USA: a major e-banking solution provider serving several thousand financial institutions has developed several different login CAPTCHA schemes.

In addition to China, Germany and USA, we also found e-banking CAPTCHAs deployed by financial institutions in other countries. These include a major bank in Switzerland (with branches in many other countries in Europe, Asia, North America and Africa), which has deployed login CAPTCHAs in its e-banking system. A Pakistani bank is also using this Swiss bank's system. Similarly, a private bank based in Latin America has also deployed login CAPTCHAs in its e-banking system, which serves its customers in Latin America, Europe, Asia, Australasia and Africa. Some Australian credit unions are also using login CAPTCHAs.

As a whole, we have found three e-banking CAPTCHA schemes for transaction verifications, one deployed by many German banks and the other two by two major Chinese banks. We found 41 e-banking CAPTCHA schemes for login. These e-banking CAPTCHA schemes involve hundreds of millions e-banking customers all around the world. In this paper, we report our successful attacks on all of these e-banking CAPTCHA schemes. We sanitized the paper to anonymize the names of all affected financial institutions and e-banking security service providers to give them the chance to amend their systems and to avoid our research results being abused by criminals. To this end, we use pseudonyms of the three e-banking CAPTCHA schemes for transaction verification: GeCapatcha refers to the e-banking CAPTCHA scheme used by German banks, ChCaptcha1 and ChCaptcha2 refer to the two used by Chinese banks.

There are also some research proposals about applications of CAPTCHAs for e-banking. In [29], Mitchell discussed the possibility of applying CAPTCHAs to e-commerce environment, where the traditional "security codes" (i.e., TANs) can be replaced by CAPTCHAs to resist automated attacks. In [30], Fischer and Herfet proposed to use CAPTCHAs for e-banking transaction verification. In [31], Szydlowski et al. proposed a CAPTCHA-based software keypad for securing web input of online transactions. In [32], a combination of CAPTCHAs and hardware security tokens is proposed to enhance e-banking security. Security and usability of these proposals remains a topic for further research.

Security analysis of e-banking CAPTCHAs is either largely unexplored or kept confidential. There are very few public reports on e-banking CAPTCHAs available. In [33], Wieser described an attack on a login CAPTCHA scheme deployed by a German bank. This attack depends on a design flaw, which has been fixed in the current deployed system.

3. CAPTCHA-BREAKING TOOLS

Despite the diversity of the e-banking CAPTCHA schemes under study, we managed to find a new set of image processing and pattern recognition tools that can break all the e-banking CAPTCHA schemes with very high success rate. Some of the tools (such as k-means clustering and morphological operations) have been widely used in the field or reported by other researchers, however, two basic tools – digital image inpainting and CW-SSIM based pattern recognition – are introduced for the first time in this paper. In the following, we briefly describe these tools, and discuss implementation issues that are common for all our attacks.

k-means layer segmentation: The first step of any attack on a CAPTCHA scheme is to extract targeted objects from the CAPTCHA image. This normally requires segmentation of the CAPTCHA image into several layers. A classic segmentation method is k-means clustering [6]. Its basic principle is to look for k cluster centroids minimizing the average distance of all points to the nearest centroid. The algorithm starts from an initial condition, and the final solution is obtained by dynamically updating the centroids.

Morphological image processing: Mathematical morphology is a theory for analysis and processing of geometrical structures [9]. It is widely used in binary images processing. The basic idea is to probe an input image with one or more pre-defined "structuring elements". There are many different morphological operations, such as dilation, erosion, opening, closing, which can be used to filter noises and refine the shape of object(s) segmented from a given image.

Line detection: Some e-banking CAPTCHAs use random lines to form a grid in order to make segmentation more difficult. To break these CAPTCHAs, we can try to detect these grid lines and then remove them. Traditionally, lines can be detected by the Hough transform [34]. In e-banking CAPTCHAs, normally grid lines have only two orientations (vertical and horizontal) and they go through the whole image. In this case, a simplified Hough transform can be used.

Digital image inpainting: This is the technique to fix missing parts in a digital image [7]. The theory behind image inpainting is to predict missing pixels from their neighbors. Some of our attacks make use of a fast image inpainting technique proposed in [8] to remove real transaction data

and replace them with fake ones in the CAPTCHA images, and to remove unwanted objects like random grid lines.

Character segmentation: For an attack on an e-banking CAPTCHA scheme, the ultimate goal is often to recognize some characters in the CAPTCHA image. This requires segmentation of each character out of the image. By applying k-means clustering or simple thresholding, we can get a layer (i.e., a binary image) containing all characters. Then, we can segment those characters out of the layer as separate connected objects. When a connected object contains more than one character, they can be split if those characters have different colors. Sometimes we also need to merge some disconnected objects into a single character (e.g. "i" and "j") according to the geometric relationship between different parts of the character. To ensure the accuracy of the character segmentation process, different kinds of morphological operations are often used to remove noises and refine the shape of segmented characters.

Character recognition: After a character is segmented, it can be further recognized. In our attacks on transaction CAPTCHAs, two training-free character recognition methods are used: CW-SSIM [11] and cross-correlation [10]. Both methods are based on template matching; i.e., they compare the input with a number of reference images (templates) to look for the best match. We avoided training based methods in this study, due to the following reasons: 1) for transaction CAPTCHA schemes it was difficult to collect a large number of images as the training set; 2) the two training-free character recognition methods work well for the CAPTCHA schemes we studied; 3) training-based methods are normally faster than template matching based methods, but the latter are easier for our proof-of-concept implementations.

Recognition error detection and correction: Due to close correlation between some reference images, the character recognition algorithm may produce erroneous results for some inputs. We developed postprocessing methods to automatically detect and correct some of these recognition errors. These methods mainly exploit the context semantics and some inherent features of the recognized characters.

To simplify implementations of our proposed attacks, we chose MATLAB as the main programming language and platform. MATLAB has a very convenient Image Processing Toolbox and an interactive programming environment. Since MATLAB is an interpreted language, its programs are significantly less efficient than those developed in compiled programming languages like C/C++. Despite this fact, most of our attacks still can run in real time. All experiments reported in this paper were done on a laptop with an Intel Core2 Duo 2.4 GHz CPU and with 2 GB memory.

4. BREAKING GECAPTCHA

GeCaptcha is a typical e-banking CAPTCHA scheme for transaction verification and being used by around 800 German banks. To use the e-banking system with GeCaptcha, each user gets a paper list of indexed TANs from the bank in advance. After the user submits a transaction request, the e-banking server sends the user a GeCaptcha image like the one shown in Fig. 2, which is a mixture of a random grid, the user's birthday, transaction data and other texts including one line with a TAN index and the transaction time. After the TAN index n (in Fig. 2, $n = 158$) is observed, the user looks for the n-th TAN in the paper list, sends the TAN back to the e-banking server for confirming the transaction.

Figure 2: A GeCaptcha image. The big digits in the background compose the user's birthday. English translation of the three text lines: Line 1 – "GeCaptcha control picture for transfer"; Line 2 – "Amount in EUR: 999,99 Bank code: 10203040 Account Nr.: 12345678"; Line 3 – "Please enter the 158th TAN".

In a GeCaptcha image, the user's birthday is used as a shared secret between the user and the e-banking server, so that the server's identity can be authenticated by the user. To defeat automated attacks, the birthday is rendered using the following operations: 1) each digit is randomly rotated; 2) the font style of each digit is randomly determined; 3) each digit is randomly located; 4) all the digits are drawn between the transaction data and the random grid, which act like decoy objects (noises) in traditional login CAPTCHA schemes. The transaction data are on top of the other layers, and it is assumed that they cannot be easily manipulated without leaving any noticeable distortion to the original GeCaptcha image.

4.1 Two Approaches to Breaking GeCaptcha

To launch a MitM attack on GeCaptcha, the attacker (i.e., the malicious program) needs to manipulate the transaction data in real time without being noticed by the user. Let us assume that the user sends transaction data TD_1 to the server, and the data are manipulated by the malicious program to $\mathrm{TD}_2 \neq \mathrm{TD}_1$. Then, the malicious program will get a GeCaptcha image with transaction data TD_2 from the server. To spoof the user, the malicious program has to manipulate the GeCaptcha image by changing TD_2 to TD_1. There are two possible approaches: 1) locate TD_2, and replace them with TD_1; 2) recognize the user's birthday and the TAN index, and forge a GeCaptcha image with TD_1.

For both approaches, the malicious program needs to first segment different objects (the transaction data, the user's birthday and the TAN index) from the GeCaptcha image. We examined histograms of many GeCaptcha images and found out that different objects correspond to different peaks in the histogram. Since we know the number of layers, the k-means clustering method [6] can be applied to segment the GeCaptcha image. For the GeCaptcha image in Fig. 2, the segmentation result is shown in Fig. 3. Based on the successful segmentation of GeCaptcha images into several layers, two automated attacks can be developed using the two approaches enumerated above.

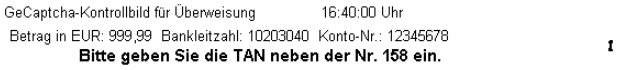
4.2 Automated Attack 1

In this attack, the malicious program achieves transaction manipulation via the following steps: Step 1) locate the text line with TD_2; Step 2) remove the text line with TD_2; Step 3) add a new line text with TD_1.

4.2.1 Step 1: Locating transaction data

The task of Step 1 is to further locate transaction data from the text layer segmented from the GeCaptcha image. The text layer contains three lines of texts: the line with transaction data, the line with TAN index, and the line with time. The order of the three lines is time-varying, so the

(a) different layers rendered with different colors



(b) the text layer

(c) the birthday layer

Figure 3: The segmentation result of the GeCaptcha image in Fig. 2.

malicious program needs a way to differentiate the line with transaction data from the other two lines.

One method to differentiate the transaction data from other texts is to recognize all the texts in the layer and then search for the keywords "Betrag in EUR", "Bankleitzahl" and "Konto-Nr.", which always appear in the line of transaction data. The recognition task can be done by an existing OCR tool due to the nearly perfect segmentation of the text layer. We tested MODI, the OCR tool included in Microsoft Office 2007, and the recognition rate is 95% for the text layer shown in Fig. 3(b). Based on such a nearly perfect recognition rate, the malicious program can easily know which line the transaction data belongs to.

While the OCR-based method works well, there is another more light-weight and robust method. It is based on the following observations: 1) the line with the TAN index is always boldfaced; 2) the line with time contains less characters than the other two lines; 3) the line with time contains a large white space. These observations imply that the average font weight (AFW) of the three lines can be very different. Let us denote the number of black pixels in a line by N_1, the number of all pixels in the bounding box of the line by N_2, the actual font size by b and the normalized font size by b_0. Then, the average font weight is defined by $\text{AFW} = (b \cdot N_1)/(b_0 \cdot N_2)$. We tested the AFWs of the three lines in 100 GeCaptcha images, and confirmed that the AFW can be used to differentiate different text lines reliably.

Based on the above finding, a more light-weight method can be easily designed to locate all the three lines. In addition to being more efficient, another advantage of the AFW-based method over the OCR-based method is that it is more robust against noise and segmentation errors.

4.2.2 *Step 2: Removing transaction data*

After locating the line with transaction data, we can try to remove the whole line by applying an image inpainting method. However, most image inpainting methods do not work well when there are too many edges around the missing parts. We found that the random grid lines and color shading of the background do introduce noticeable distortions. Figure 4 shows the results of applying the image inpainting method in [8] to the GeCaptcha image in Fig. 2 by taking the line with transaction data as the mask of the to-be-filled region. We can see some noticeable distortions such as broken grid lines.[3] We tried two other image inpainting methods [35, 36] and got similar results.

Figure 4: The result of removing transaction data by applying the image inpainting method in [8].

To overcome the problem, we have to handle pixels on the grid lines separately: they should be predicted from closest (not necessarily neighboring) pixels on grid lines and should not be used for predicting pixels that do not lie on grid lines. Based on this idea, we extended the fast image inpainting method proposed in [8]. The extended method needs to know where the grid lines are. As we described in Sec. 3, these (horizontal and vertical) grid lines can be detected by a simplified Hough transform. After the grid lines are localized, pixels on the random grid can be handled differently, so that no visible distortion will occur on and around grid lines. Figure 5 shows the result of the extended inpainting method. Comparing Fig. 5 with Fig. 4, we can see that our proposed inpainting method, while having the same level of complexity, creates significantly lesser distortion than general-purpose inpainting methods.

Figure 5: The result of applying the extended inpainting method to the GeCaptcha image in Fig. 2.

4.2.3 *Step 3: Adding user-expected transaction data*

After removing the transaction data TD_2, it is trivial to add the user-expected transaction data TD_1 to the vacant place in the GeCaptcha image. Figure 6 shows a forged GeCaptcha image by changing the transaction data to "Betrag in EUR: 1,00 Bankleitzahl: 18635402 Konto-Nr.: 1211855". We tested the image inpainting based attack on 100 GeCaptcha images collected from real user accounts and no visual distortion is observed in any forged GeCaptcha image, thus leading to the ideal success rate of 100%.

Figure 6: A forged GeCaptcha image from the GeCaptcha image in Fig. 2.

4.3 Automated Attack 2

The image inpainting based attack described above is blind, in the sense that it does not depend on a recognition task. However, if we can recognize the user's birthday and the TAN index embedded in the GeCaptcha image, a second attack can be developed. An additional advantage of the second attack is that the attacker can get the user's birthday, which is the user's private information that plays a key role in some backup authentication systems. The second attack consists of the following two stages.

Stage 1 – birthday recognition: The malicious program collects a number of GeCaptcha images, and tries to recognize the user's birthday. This stage is completely offline.

[3] Although users are not always sensitive to those subtle distortions, they can be easily trained to be more careful.

Stage 2 – transaction manipulation: For a new transaction request received from the user, the malicious program manipulates the transaction data, then locates (probably also recognizes) the TAN index from the server-generated GeCaptcha image, and finally sends a forged GeCaptcha image with the user's transaction data back to the user. This stage has to be done online in real time.

4.3.1 Stage 1: Offline birthday recognition

As shown in Sec. 4.1, the image segmentation process can produce a segmented layer with birthday. This layer can be further segmented to extract each birthday digit. Some morphological operations are needed to filter small objects and noises, and to refine the shapes of segmented birthday digits. Figure 7 shows the digits segmented from the birthday layer shown in Fig. 3(c).

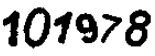


Figure 7: The eight birthday digits segmented from the birthday layer shown in Fig. 3c.

Since the birthday digits are normally rotated and the segmentation result is not always perfect, OCR tools do not work very well. Instead, we chose to use a training-free algorithm CW-SSIM [11] to recognize these digits. CW-SSIM denotes "complex wavelet based structural similarity", which is a full-reference image quality assessment (IQA) algorithm invariant to translation and small scaling/rotation. In [11], it was demonstrated how CW-SSIM can be used to achieve robust and highly accurate digit recognition.

To use CW-SSIM, we need a database of reference images of the to-be-recognized digits. We used a database with three rotation angles (0 and ± 15 degrees) and two different font styles (boldfaced, boldfaced italic). As a whole, there are $10 \times 3 \times 2 = 60$ reference images. Based on such a database, the birthday in Fig. 7 can be successfully recognized. For 100 GeCaptcha images, the success rate is 91%.

The success rate can be further improved by using image inpainting. The idea is to remove the whole text layer and the grid lines, which normally leads to a better segmentation result of the birthday layer and thus also the birthday digits. Figure 8 shows the result of removing all those unwanted objects from the GeCaptcha image Fig. 2. One can see that in the inpainted image the birthday becomes more salient in the background. For the simplified GeCaptcha image, a 3-means clustering process is used to extract the birthday for recognition. With the improved method, the success rate of birthday recognition becomes 100%.

Figure 8: The inpainting result of the GeCaptcha image in Fig 2, by removing all unwanted objects.

4.3.2 Stage 2: Online transaction manipulation

Once the user's birthday is broken, the malicious program can start manipulating transaction data and forging GeCaptcha images. To do so, the malicious program needs to locate the line with TAN index in the server-generated GeCaptcha image because the server expects a specific TAN for confirming the (manipulated) transaction. As we described in Sec. 4.2.1, we can segment the line with TAN index from the text layer by using the AFW-based method.

After extracting the line with TAN index, the malicious program can synthesize a fake GeCaptcha image from the following known information: the user's birthday, the line with TAN index, the original transaction data TD_1 and the current time. We developed an image generation engine to do this task. Figure 9 shows an example of the forged GeCaptcha image by our image generation engine.

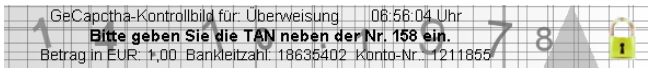


Figure 9: A GeCaptcha image synthesized from the image in Fig. 2 after the birthday is recognized.

4.4 Human-involved semi-automated attack

In Automated Attack 2, the first stage is to recognize the user's birthday, which is done offline. Instead of building its own recognizer, the malicious program can also send the segmented birthday to a human solver to recognize the birthday. Such an attack will be useful if GeCaptcha is enhanced to make the birthday recognition task very difficult.

The human solver is not necessarily the attacker himself. The malicious program can create a CAPTCHA image from the segmented birthday and send it to a web site under its control as a challenge for login, which will be solved by a visitor of the web site. After the malicious program obtains the recognized birthday from a human observer, the second stage of Automated Attack 2 can be launched as usual.

A salient feature of this attack is that the human solver is needed only once and the whole process afterwards is fully automated. This explains why we call it "human-involved semi-automated attack".

4.5 Efficiency of the proposed attacks

For 100 test images, the average running time of the inpainting based attack is around 250 ms, and that of Stage 2 (online transaction manipulation) of the recognition based attack is around 190 ms. The running time starts from reading the real image from hard disk and ends with storage of the forged image on hard disk.

Stage 1 (the birthday recognition part) of the birthday recognition attack is relatively slow because the CW-SSIM values have to be calculated for all the birthday digits and all the reference images. The average time of birthday recognition is around 5 seconds. The efficiency problem is not a big issue because: 1) the recognition stage runs offline; 2) the recognition can be made faster by replacing CW-SSIM with a training-based recognizer; 3) the MATLAB code we used has significant room for further optimization.

5. BREAKING CHCAPTCHA1 AND CHCAPTCHA2

ChCaptcha1 and ChCaptcha2 are e-banking CAPTCHA schemes used by two major banks in China. The two schemes are very similar to GeCaptcha, so they can be broken by generalizing the attacks described in the above section.

5.1 Breaking ChCaptcha1

ChCaptcha1 is similar to GeCaptcha, but with three main differences. First, there is no paper list of TANs issued to each user. Instead, four digits in the receiver's account number are randomly selected and rendered in color. The user is asked to input these four digits as a transaction-dependent TAN. Second, there is no secret shared between the user and the server for server authentication. Third, there are no random grid lines. Figure 10 shows a ChCaptcha1 image we collected from a real bank account. In the background of the ChCaptcha1 image, multiple copies of the bank's logo are embedded. We replace these logos by white disks with gray edges to avoid exposing the bank's identity.

Figure 10: A ChCaptcha1 image. English translation of the texts: Line 1 – "receiver's account"; Line 2 – "receiver's name"; Line 3 – "TAN Please input the big red digits in receiver's account".

In ChCaptcha1, the TAN is embedded into the CAPTCHA image, so a recognition based attack is able to break the CAPTCHA scheme. The attack is similar to Automated Attack 2 on GeCaptcha: layer segmentation → transaction data localization → TAN digit segmentation → TAN digit recognition. The segmentation results of the ChCaptcha1 scheme is nearly perfect and the TAN digits are not rotated, so the simpler correlation based method can be used for recognition. We tested the recognition based attack on 100 ChCaptcha1 images and achieved a success rate of 100%. The average running time of the attack is less than 150 ms.

5.2 Breaking ChCaptcha2

ChCaptcha2 does not depend on a paper list of TANs, either. A 5-digit TAN is dynamically generated and embedded into the CAPTCHA image like the user's birthday in a GeCaptcha image. Different from the ChCaptcha1 scheme, the ChCaptcha2 TAN is not transaction dependent. There are no random grid lines, either. Figure 11 shows a ChCaptcha2 image we collected from a real bank account.

Figure 11: A ChCaptcha2 image. English translation of the texts: Line 1 – "Attention! Please check the following information carefully"; Line 2 – "receiver's account"; Line 3 – "receiver's name"; Line 4 – "amount".

Compared with ChCaptcha1, ChCaptcha2 is more similar to GeCaptcha. The two attacks on GeCaptcha can both be generalized. The processes are nearly the same as those on GeCaptcha, except that the color information is also used for k-means clustering. We tested both attacks on 103 ChCaptcha2 images, and the success rate is 100%. The efficiency of the recognition based attack is relatively low, with an average running time of about 6-7 seconds. Note, however, that the malicious program does not need to respond to the server in real time as the CAPTCHA images are supposed to be solved by human users who can be very slow. On the other hand, the malicious program can still interact with the user in real time because it does not need to wait for the recognition result to forge a CAPTCHA image.

6. BREAKING LOGIN CAPTCHAS

In addition to the three e-banking CAPTCHA schemes for transaction verification, we found 41 e-banking CAPTCHA schemes for login. Our study on these e-banking CAPTCHA schemes is alarming: *all* of them are insecure against automated segmentation attacks. Some of them are designed in such a naive way that the segmentation information of the CAPTCHA images have been fully or partially encoded in the images themselves. Since character recognition is not difficult if the characters have been well segmented [19, 21], the success of our segmentation attacks have shown that all of these e-banking login CAPTCHA schemes are not secure.

Our segmentation attacks on the 41 e-banking CAPTCHA schemes for login are based on the same set of CAPTCHA-breaking tools described in Sec. 3, so we do not repeat the detail about how each login CAPTCHA scheme is broken. Instead, in Table 1 we show segmentation results of some selected login CAPTCHA schemes[4]. We also list the tool(s) we used and weakness(es) we exploited in our attacks. Character segmentation is used for all schemes, so it is not listed in the table to save space. For each e-banking login CAPTCHA scheme, we have tested the segmentation attack on at least 60 sample images to estimate the success rates.

7. MORE DISCUSSIONS

Our attacks on e-banking CAPTCHAs raise the question of whether financial institutions should continue to use CAPTCHAs for their e-banking services or they should leave them for more secure solutions. That is, the following question needs to be answered: *can we enhance the broken e-banking CAPTCHA schemes so that they are immune to the proposed attacks?* In this section, we first take a look at the case of e-banking CAPTCHAs for transaction verification and then discuss all e-banking CAPTCHAs as a whole.

7.1 Can transaction CAPTCHAs be enhanced?

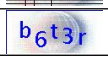
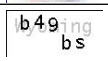
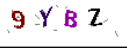
Due to the similarity of the three transaction e-banking CAPTCHA schemes under study, in this subsection we will focus on GeCaptcha to ease our discussion.

One simple improvement is to compress the image with a lossy algorithm like JPEG, in the hope that the boundaries between different objects are blurred so that the attacks become difficult. Unfortunately, our attacks can be easily enhanced to tolerate lossy compression by adding an additional noise filter. Our experiments showed that the inpainting based attack works even when the lowest quality factor of JPEG compression is used. As a consequence, lossy compression cannot enhance the security of GeCaptcha.

There are some other possible improvements: replacing the random grid lines by random curves, balancing the three text lines so that they have similar AFWs, changing the birthday to a different form such as a number of secret

[4]Due to the page limit, we cannot list all login CAPTCHA schemes here. A full list is available at `http://www.hooklee.com/default.asp?t=eBankingCAPTCHAs`.

Table 1: Selected e-banking login CAPTCHA schemes we studied, with results of our segmentation attacks.

Financial institution(s)/e-banking login CAPTCHA scheme	*CAPTCHA image(s)*	*Segmentation result(s)*	*Tool(s) used, Weakness(es) exploited*	*Success rate*
13 German banks		534261	3-means clustering, morphological operations	100%
Hundreds other German banks		Q4B254	2-means clustering, line detection, image inpainting	100%
A Swiss bank with branches in Europe, Asia, North America and Africa		49575	2-means clustering	100%
A bank based in Latin America with branches in Europe, Asia, Australasia and Africa		31289	2-means clustering	100%
US e-banking CAPTCHA 1		454e	2/3-means clustering, line detection, image inpainting	100%
US e-banking CAPTCHA 2		b6t3r	3-means clustering	100%
US e-banking CAPTCHA 3		b49bs	3-means clustering	100%
US e-banking CAPTCHA 4		3264	2/3-means clustering	100%
Three CUs in Australia		bQUM	3-means clustering, morphological operations	99.5%
Chinese e-banking CAPTCHA 1		3571	3-means clustering	100%
Chinese e-banking CAPTCHA 2		9YBZ	2-means clustering, image inpainting	100%
Chinese e-banking CAPTCHA 3		88071	4-means clustering, morphological opening	100%
Chinese e-banking CAPTCHA 4		WLAV3	morphological cleaning, character intensity < 120	100%
Chinese e-banking CAPTCHA 5		bYjg	3-means clustering, morphological cleaning	98.3%
Chinese e-banking CAPTCHA 6		3m9nxn	grayscale foreground vs. colored noises	100%
Chinese e-banking CAPTCHA 7		smib	2-means clustering	100%
Chinese e-banking CAPTCHA 8		xjva3	3-means clustering	100%
Chinese e-banking CAPTCHA 9		MKJq	2/3-means clustering	100%
Chinese e-banking CAPTCHA 10		mj4k	2-means clustering, morphological operations	95.1%

icons, changing the order of different layers, etc. Unfortunately, none of these improvements can resist the two proposed automated attacks simultaneously. Even if we combine all of them, the human-involved semi-automated attack still works, as long as the text layer can be extracted.

A more effective improvement is to change the gray scale of different objects in the GeCaptcha image so that they overlap with each other in the histogram. This will make k-means clustering fail. For an enhanced GeCaptcha image shown in Fig. 12, none of our proposed attacks works.

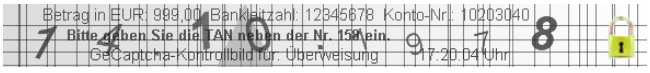
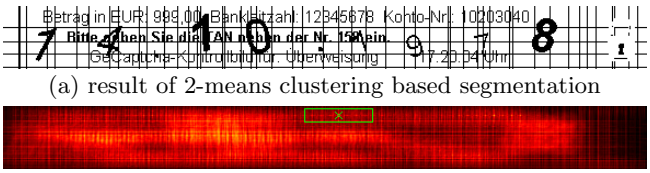


Figure 12: An enhanced GeCaptcha image in which different foreground layers have similar gray values.

Unfortunately, the failure of our proposed attacks does not mean that more advanced attacks cannot be developed. In fact, based on an idea similar to the generalized Hough transform [37], we can develop a more advanced attack. The basic idea is as follows: since the malicious program knows many texts embedded in a GeCaptcha image and the types of distortions applied to these texts, it can build a database of shape templates of these texts according to all the possible rendering configurations. Then, the malicious program tries to search all shape templates in the GeCaptcha image to find the one leading to the best match at a specific location. This will tell where the target texts and their contextual texts are, which can then be segmented and manipulated or recognized. Here, we can show an example for the enhanced GeCaptcha image in Fig. 12. Let us assume that the malicious program wants to manipulate the receiver's account number. Since the malicious program knows the receiver's account number, it can create a number of templates and search for them in the GeCaptcha image. The maximal correlation will show the correct location of the receiver's account number. A 2-means clustering process can be performed before the searching process starts so that only the foreground will be matched. Since the background and the foreground have to maintain a considerable contrast to make the foreground visible, the 2-means clustering should always work very well. Figure 13 shows the result of searching for the receiver's account number. We can see that it is exactly localized, and hence transaction manipulation becomes easy.

The template searching based attack is very powerful. It



(a) result of 2-means clustering based segmentation

(b) correlation map

Figure 13: The result of searching for the account number "12345678" in Fig. 12. The green rectangle shows the location with the maximal correlation.

works well for transaction e-banking CAPTCHAs because many characters known to the malicious program (e.g., transaction data/time) have to be embedded into the CAPTCHA image. To improve security against such an attack, we must increase the number of distinct text rendering and distortion methods so that the searching process becomes extremely slow and/or storing all templates becomes impossible. But this will increase the complexity of the CAPTCHA scheme itself. It is also doubtful if there are enough rendering parameters and distortions because: 1) both machines and humans are not sensitive to small changes of rendered texts; 2) distortions have to remain light to maintain visibility of the texts and usability of the CAPTCHA scheme.

7.2 Are CAPTCHAs good for e-banking at all?

While it seems difficult to improve the security of transaction CAPTCHAs, we still have the last (somewhat circular reasoning) resort: to render all texts as strong CAPTCHAs. This is also the way to enhance e-banking CAPTCHAs for login. Here, "strong" means that *any* automated attacks based on the state-of-the-art techniques is impractical. Then, the question becomes if such strong CAPTCHAs do exist. This question is difficult to answer conclusively as an accurate definition of hard AI problems does not exist. Moreover, unavailability of (publicly) known attacks on a specific CAPTCHA scheme does not mean that such attacks do not exist. For instance, Google's reCAPTCHA uses words that cannot be recognized by the state-of-art OCR tools to generate strong CAPTCHA images, which is believed to be secure due to the creative way of CAPTCHA image generation. However, recently some automated attacks on reCAPTCHA were reported [38]. Although reCAPTCHA can be updated, the attacks will also evolve and new attacks may emerge.

In addition to the security problem of CAPTCHAs, there is also a well-known tradeoff between security and usability [39]. To make a CAPTCHA scheme more secure, often usability has to be compromised, and vice versa. For e-banking systems, this security-usability tradeoff is more critical. This is because customers who have trouble with strong CAPTCHAs may complain and even switch to other financial institutions. We believe this is part of the reason why many financial institutions have not deployed CAPTCHAs or have deployed less secure (but more usable) CAPTCHAs.

Relying on CAPTCHAs for e-banking has a salient drawback related to the tradeoff between security and usability: financial institutions have to be prepared to patch their system at *any* time since new attacks may appear at *any* time. This will inevitably increase maintenance costs. Financial institutions may choose to patch their e-banking systems less frequently, thus leaving security holes in their systems.

The nature of CAPTCHAs implies that they are vulnerable to human-involved attacks. Compared to other applications of CAPTCHAs, attackers will have more incentives to employ cheap labor to solve e-banking CAPTCHAs [40]. Although the attacker has to ensure real-time response in some cases, this can be achieved if the attacker can exploit the user base of a popular web site. In case the attacker can infect a large number of computers, which has already been happening in today's Internet, the chance to be successful for at least one victim can be practically high. Since 2007, some malware has been found to use this strategy [41].

We can also compare e-banking CAPTCHAs with traditional schemes and from an economic perspective. In traditional applications of CAPTCHAs, breaking a CAPTCHA scheme leads to only abuse of the resources protected by the CAPTCHA scheme. However, for e-banking systems, breaking a CAPTCHA scheme can cause a potentially huge loss for both users and banks. As a whole, we have the feeling that CAPTCHAs may be incapable of protecting e-banking systems, due to the higher security requirements. In [42], Jakobsson expressed the same concern and proposed an alternative solution called "CAPTCHA-free throttling".

Based on the above discussion, we call for cautions in deploying e-banking CAPTCHAs. For financial institutions relying only on CAPTCHAs, we suggest moving to alternative solutions or at least combining CAPTCHAs with other solutions. Among all alternative solutions, we feel that hardware security tokens are more promising. Not all hardware based solutions can resist MitM attacks, but at least some can, using transaction-dependent TANs, encrypted channels, and/or trusted display/keypad. For instance, if a hardware token is equipped with a trusted display and can sign the transaction data, the user can ensure "what she sees is what she signs", thus rendering MitM attacks impossible. Of course, hardware based solutions are not perfect, either. Their main disadvantage is that either the financial institution or the customer has to bear the additional costs.

8. CONCLUSIONS

This paper reports a comprehensive study on e-banking CAPTCHA schemes, including three for transaction verification and 41 schemes for login. We propose a new set of image processing and pattern recognition techniques to break all the e-banking CAPTCHA schemes with a success rate equal to or close to 100%. We have also shown that it is a nontrivial task to enhance e-banking CAPTCHA schemes to ensure both security and usability. We thus raise the question if financial institutions should rely on e-banking CAPTCHAs at all. Our opinion is that e-banking CAPTCHAs are better replaced by other alternative solutions such as those based on hardware security tokens.

9. ACKNOWLEDGMENTS

Shujun Li was supported by the Zukunftskolleg ("Future College") of the University of Konstanz, Germany, which is part of the "Excellence Initiative" Program of the DFG (German Research Foundation). The work of S. Amier Haider Shah, M. Asad Usman Khan and Syed Ali Khayam was partially supported by the Pakistan National ICT R&D Fund.

10. REFERENCES

[1] American Bankers Association. Consumers prefer online banking. `http://www.aba.com/Press+Room/`

092109ConsumerSurveyPBM.htm, 2009.

[2] Bank Austria. mobileTAN information. http://www.bankaustria.at/de/19741.html, 2007.

[3] Cronto Limited. Cronto's visual cryptogram. http://www.cronto.com/visual_cryptogram.htm, 2008.

[4] Volksbank Rhein-Ruhr eG. Bankgeschäfte online abwickeln: Mit Sm@rtTAN optic bequem und sicher im Netz. http://www.voba-rhein-ruhr.de/privatkunden/ebank/SMTop.html, 2009.

[5] T. Weigold, T. Kramp, R. Hermann, F. Höring, P. Buhler, and M. Baentsch. The Zurich Trusted Information Channel – an efficient defence against man-in-the-middle and malicious software attacks. In *TRUST'2008*, pages 75–91.

[6] G. A. F. Seber. *Multivariate Observations*. John Wiley & Sons, Inc., 2004.

[7] M. Bertalmio, G. Sapiro, V. Caselles, and C. Ballester. Image inpainting. In *SIGGRAPH'2000*, pages 417–424.

[8] M. M. Oliveira, B. Bowen, R. McKenna, and Y.-S. Chang. Fast digital image inpainting. In *IASTED VII'2001*, pages 261–266.

[9] F. Y. Shin. *Image Processing and Mathematical Morphology*. CRC, 2009.

[10] S. J. Orfanidis. *Optimum Signal Processing*. 2 edition, 2007. http://www.ece.rutgers.edu/~orfanidi/osp2e.

[11] Z. Wang and E. P. Simoncelli. Translation insensitive image similarity in complex wavelet domain. In *ICASSP'2005*, pages 573–576.

[12] L. von Ahn, M. Blum, N. J. Hopper, and J. Langford. CAPTCHA: Using hard AI problems for security. In *EUROCRYPT'2003*, pages 294–311.

[13] J. Elson, J. R. Douceur, J. Howell, and J. Saul. Asirra: A CAPTCHA that exploits interest-aligned manual image categorization. In *CCS'2007*, pages 366–374.

[14] A. Basso and F. Bergadano. Anti-bot strategies based on human interactive proofs. In *Handbook of Information and Communication Security*, chapter 15, pages 273–291. Springer, 2010.

[15] G. Mori and J. Malik. Recognizing objects in adversarial clutter: Breaking a visual CAPTCHA. In *CVPR'2003*, pages 134–141.

[16] G. Moy, N. Jones, C. Harkless, and R. Potter. Distortion estimation techniques in solving visual CAPTCHAs. In *CVPR'2004*, pages 23–28.

[17] K. Chellapilla and P. Y. Simard. Using machine learning to break visual Human Interaction Proofs (HIPs). In *NIPS'2004*, pages 265–272, 2005.

[18] S. Hocevar. PWNtcha: Pretend we're not a Turing computer but a human antagonist. http://caca.zoy.org/wiki/PWNtcha, 2004.

[19] K. Chellapilla, K. Larson, P. Simard, and M. Czerwinski. Computers beat humans at single character recognition in reading based human interaction proofs (HIPs). In *CEAS'2005*.

[20] J. Yan and A. S. El Ahmad. Breaking visual CAPTCHAs with naïve pattern recognition algorithms. In *ACSAC'2007*, pages 279–291.

[21] J. Yan and A. S. El Ahmad. A low-cost attack on a Microsoft CAPTCHA. In *CCS'2008*, pages 543–554.

[22] P. Golle. Machine learning attacks against the Asirra CAPTCHA. In *CCS'2008*, pages 535–542.

[23] J. Tam, J. Simsa, S. Hyde, and L. von Ahn. Breaking audio CAPTCHAs. In *NIPS'2008*, pages 1625–1632, 2009.

[24] E. Bursztein and S. Bethard. Decaptcha: Breaking 75% of eBay audio CAPTCHAs. In *WOOT'2009*.

[25] C. J. Hernandez-Castro and A. Ribagorda. Pitfalls in CAPTCHA design and implementation: The Math CAPTCHA, a case study. *Computers & Security*, 29(1):141–157, 2010.

[26] A. Hindle, M. W. Godfrey, and R. C. Holt. Reverse engineering CAPTCHAs. In *WCRE'2009*, pages 59–68.

[27] C. J. Hernandez-Castro and A. Ribagorda. Remotely telling humans and computers apart: An unsolved problem. In *iNetSec'2009*.

[28] B. Pinkas and T. Sander. Securing passwords against dictionary attacks. In *CCS'2002*, pages 161–170.

[29] C. J. Mitchell. Using human interactive proofs to secure human-machine interactions via untrusted intermediaries. In *Security Protocols'2006*, pages 164–170, 2009.

[30] I. Fischer and T. Herfet. Visual CAPTCHAs for document authentication. In *MMSP'2006*, pages 471–474.

[31] M. Szydlowski, C. Kruegel, and E. Kirda. Secure input for Web applications. In *ACSAC'2007*, pages 375–384.

[32] D. J. Steeves and M. W. Snyder. Secure online transactions using a CAPTCHA image as a watermark. US Patent 2007/0005500.

[33] W. Wieser. Captcha recognition via averaging. http://www.triplespark.net/misc/captcha, 2007.

[34] R. C. Gonzalez and R. E. Woods. *Digital Image Processing*. Prentice Hall, 2008.

[35] A. Criminisi, P. Pérez, and K. Toyama. Object removal by exemplar-based inpainting. In *CVPR'2003*, pages 721–728.

[36] P. Getreuer. tvreg: Variational imaging methods for denoising, deconvolution, inpainting, and segmentation. http://www.math.ucla.edu/~getreuer/tvreg.html, 2009.

[37] D. H. Ballard. Generalizing the Hough transform to detect arbitrary shapes. *Pattern Recognition*, 13(2):111–122, 1981.

[38] J. Wilkins. Strong CAPTCHA guidelines: v1.2. http://bitland.net/captcha.pdf, December 2009.

[39] J. Yan and A. S. El Ahmad. Usability of CAPTCHAs or usability issues in CAPTCHA design. In *SOUPS'2008*, pages 44–52.

[40] M. Motoyama, K. Levchenko, C. Kanich, D. McCoy, G. M. Voelker, and S. Savage. Re: CAPTCHAs – Understanding CAPTCHA-solving services in an economic context. In *USENIX Security'2010*.

[41] BBC News. PC stripper helps spam to spread. http://news.bbc.co.uk/2/hi/technology/7067962.stm, 2007.

[42] M. Jakobsson. CAPTCHA-free throttling. In *AISec'2009*, pages 15–21.

FIRM: Capability-based Inline Mediation of Flash Behaviors

Zhou Li, XiaoFeng Wang
Indiana University, Bloomington
{lizho,xw7}@indiana.edu

ABSTRACT

The wide use of Flash technologies makes the security risks posed by Flash content an increasingly serious issue. Such risks cannot be effectively addressed by the Flash player, which either completely blocks Flash content's access to web resources or grants it unconstrained access. Efforts to mitigate this threat have to face the practical challenges that Adobe Flash player is closed source, and any changes to it need to be distributed to a large number of web clients. We demonstrate in this paper, however, that it is completely feasible to avoid these hurdles while still achieving fine-grained control of the interactions between Flash content and its hosting page. Our solution is *FIRM*, a system that embeds an inline reference monitor (IRM) within the web page hosting Flash content. The IRM effectively mediates the interactions between the content and DOM objects, and those between different Flash applications, using the capability tokens assigned by the web designer. FIRM can effectively protect the integrity of its IRM and the confidentiality of capability tokens. It can be deployed without making any changes to browsers. Our evaluation based upon real-world web applications and Flash applications demonstrates that FIRM effectively protects valuable user information and incurs small overhead.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Unauthorized access

General Terms

Security

Keywords

Adobe Flash, Cross-site scripting, Inline Reference Monitor

1. INTRODUCTION

Flash, a multimedia platform first introduced in 1996, has been extensively used today to deliver dynamic web contents, including animations, advertisements, movies and others. Websites such as YouTube serve hundreds of millions of videos every day. Popular portals, such as CNN, Yahoo, etc., broadcast news and host advertisements through Flash contents. The pervasiveness of Flash, however, brings in new security and privacy concerns. Adobe Flash player offers interfaces for a Flash application (Shock-Wave-Flash file or SWF) to operate on the DOM (document object model) objects of its hosting page, other SWF files and file systems. Through ActionScript functions such as `getURL`, Flash content can even inject JavaScript code into web content. Without proper control, it is conceivable that malicious Flash code can wreak such havoc as stealing sensitive information (e.g., cookies, passwords) and modifying high-integrity data (e.g., account balances). Such threats can also come from legitimate yet vulnerable Flash content: as discovered recently [29, 34, 18], a large number of existing Flash applications contain serious security flaws that can be exploited to launch attacks like cross-site scripting (XSS), cross-site request forgery (XSRF), and others. As an example, last year, an XSS flaw was found in the Flash content hosted by the SSL e-banking site of Marfin Egnatia Bank [23], through which an attacker can inject malicious scripts to steal credentials from the bank's customers.

Adobe Flash player provides security mechanisms to control interactions between Flash code and its hosting page: web developers can determine whether a Flash application should be allowed to operate on DOM objects, for example, through script injection. Specifically, when embedding the Flash application, the web developer declares a property named `allowScriptAccess` with one of the values: `always`(scripting allowed), `sameOrigin`(scripting allowed only when the hosting page and Flash code are from same origin) or `never`(scripting prohibited). Once scripting is allowed, the injected code automatically acquires unlimited access to DOM objects. Adobe also controls the interactions between different SWFs according to Same Origin Policy [38]. SWF files and other files are grouped into sandboxes by virtue of the domains they originate from. A Flash application can directly access the resources within its sandbox but needs mediation for any cross-domain access. This security control, again, is black-and-white, which either grants a Flash application and other SWF files it downloads full access or completely denies their access. Such a treatment turns out to be too coarse-grained to be useful for real-world Flash serving websites. Many legitimate Flash applications need script injection. Examples include CNN [5] that lets Flash advertisements utilize JavaScript to enrich their visual effects, and Yahoo [15] that allows such advertisements to track user clicks and profile through scripts. Overly restricting Flash/DOM interactions can significantly reduce the utility of Flash and is often suggested against [17]. As

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

a result, many websites are forced to give Flash code unlimited access, which exposes valuable information assets on the web client to the threat of malicious or vulnerable Flash content.

Our approach. A practical solution to this problem is by no means trivial. Modifying the security mechanism of Adobe Flash player is not feasible, as the software is closed source. Even if this can be done, deployment of a new mechanism requires changes to every client's browser, a slow and painful process. In this paper, we present an effective and convenient alternative. Our techniques allow Flash hosting sites to offer immediate protection to their customers' valuable web contents. This is achieved through an *Inline Reference Monitor (IRM)* system, called *FIRM*, that embeds an access control mechanism entirely into web pages. Through FIRM, the website designer can assign *capability* tokens to different Flash applications the site hosts. Each token is associated with a set of security policies that specify an application's privileges over web contents, including DOM objects and other SWF files. Such a policy is enforced by an *IRM* that wraps both ActionScript functions within SWF files and DOM functions. As an example, consider a Flash advertisement that needs to run a script to track viewers' clicks. Our approach first analyzes the binary code of the Flash, instruments it with a *Flash wrapper* and also grants it a capability token. The new Flash code is served within a web page that also includes a *DOM wrapper* and a set of security policies. When a user is browsing the page, the *Flash wrapper* intercepts the `getURL` call from the advertisement and works with the DOM wrapper to decide whether to let the call proceed based upon the capability.

FIRM offers flexible and fine-grained control over Flash / DOM and Inter-Flash access. It can be completely embedded into a web page by web developers, and therefore avoids any browser-side changes, which makes its deployment instant. Our technique is also reliable: the design of FIRM prevents unauthorized dynamic contents, such as scripts and SWF files, from stealing authorized parties' capability tokens or modifying the IRM and its policy data. We also built a tool for automatic analysis and instrumentation of Flash code. We evaluated our approach on phpBB [11], WordPress [14] and Drupal [6], 3 extremely popular web-design systems, and 9 real-world Flash applications. Our study shows that FIRM effectively mediates Flash behaviors, incurs small overheads and is convenient to use.

Contributions. We summarize the contributions of the paper as follows:

• *Novel Flash mediation techniques.* To the best of our knowledge, FIRM is among the first attempts to enforce inline mediation of Flash/DOM interactions. This is achieved through a novel capability mechanism, which employs randomized, unpredictable tokens to differentiate the access requests that come directly or indirectly (through JavaScript) from the Flash applications with different privileges. Our mechanism can also effectively protect itself from malicious web contents, and automatically instrument Flash applications and web pages.

• *Capability-based inlined mediation of JavaScripts.* The behaviors of Flash applications cannot be effectively mediated without proper control of the scripts they spawn. Different from the prior approach [36] that uses the same set of policies to control all scripts within a web page, FIRM can enforce different policies on different scripts, according to the privileges associated with their capability tokens. This finer-grained access control mechanism is enforced with the collaborations between the DOM wrapper and the Flash wrapper.

• *Implementation and Evaluation.* We implemented a prototype of FIRM and evaluated it on popular web applications and real-world Flash. The outcomes of this study demonstrate the efficacy of our techniques.

Roadmap. The rest of the paper is organized as follows. Section 2 introduces the attack techniques in Flash malware. Section 3 surveys the design of FIRM and the adversary model. Section 4 elaborates the techniques for Flash inline mediation. Section 5 documents our Flash analysis and instrumentation techniques. Section 6 reports our experimental study. Section 7 discusses the limitations of our techniques and future research. Section 8 compares our approach with prior work, and Section 9 concludes the paper.

2. FLASH THREATS

In this section, we briefly review the threats FIRM is designed to mitigate. These threats come from malicious or vulnerable Flash applications.

Illegitimate operations on DOM objects. Through Adobe Flash player, Flash code can directly access DOM objects. Such access, if unmediated, could cause leak of sensitive user data (e.g., cookie) as well as unexpected change of browser behavior. A prominent example is the re-direction attack [21]: a malicious Flash application can redirect the user's browser to a malicious website, where subsequent attacks like drive-by download, phishing, etc. can happen.

Script injection. A Flash application can inject JavaScript code into its hosting web page through ActionScript calls such as `getURL`, which can be exploited to launch an XSS attack. This threat has been widely reported [18, 8, 34]. An example in Figure 1 shows how it happens through vulnerable Flash code. Flash applications (particularly advertisements) often use the variable `clickTag` to receive URLs from its hosting page and redirect the user to these links. This feature can be exploited by the attacker, who can create a link that invokes the Flash code with `clickTag` involving JavaScript code. Once the victim clicks on that link, the Flash injects the script into her web page, using the origin of its hosting page. To eliminate such a threat, a website needs to detect and fix the flaws in every SWF file it stores or links to. This can introduce considerable overhead, particularly for the websites such as Yahoo that host hundreds of thousands of Flash advertisements from other sites. Moreover, malicious scripts can also be injected by malicious Flash applications, which allow them to indirectly access the victim's data.

Figure 1: Vulnerable Flash with XSS flaw

Other threats. Though less known, other attack avenues do exist during Flash/DOM interactions and inter-Flash interactions. For example, once a Flash application shares a single function to another Flash, the Flash player automatically exposes all its functions to the latter. As another example, Flash can export function interfaces to its hosting page, which can be invoked by any script in the page. This channel can be used to bypass the security policy enforced by the Flash player: consider that a Flash application is not

allowed to be touched by another Flash but needs to share its functions to the hosting page; the latter can then inject scripts into the page to gain access to those functions.

Assumptions. Our approach protects the user's valuable web content, such as cookies, passwords and account numbers, from unauthorized access by malicious Flash code or the vulnerable Flash exploited by the adversary. We assume that the website hosting SWF files is not under the control of the adversary and implements FIRM correctly. Also, though FIRM can protect itself against malicious web contents, it is not resilient to a compromised browser or operating system. For example, a malware plug-in can certainly bypass the mediation of our IRM. Such a threat is out of the scope of this work. Finally, FIRM is designed to regulate Flash/DOM and inter-Flash interactions. Other Flash-related attacks, like filling web surfer's clipboard with malicious hyperlinks [2], are not the focus of our approach and left to our future research.

3. OVERVIEW

FIRM includes a *Flash wrapper* for mediating Flash actions and a *DOM wrapper* for controlling the activities of the scripts. Both wrappers interact with a *capability manager* that bootstraps the reference monitor with randomly generated capability tokens, and maintains a *policy base* to map these tokens to the security policies set by the web designer. The wrappers and the capability manager constitute the IRM part of FIRM. The other FIRM component is the tool that automatically embeds the Flash wrapper into a SWF file through analyzing and instrumenting its binary code. Figure 2 illustrates this design.

3.1 Design

The website that uses FIRM first embeds our IRM into the web page that needs protection. Whenever the page is requested by a web client, the site automatically parameterizes it with a set of randomized capability tokens, which are associated with pre-determined security policies (Section 4.1). These policies grant different privileges to different SWF files, as determined by the web designer *a priori*. The token of each Flash application is checked by the wrappers against the policies to control the Flash's access to the web contents (Section 4.2). This idea can be explained with an example in Figure 3, which describes a Flash advertisement with FIRM instrumentation. In the example, FIRM permits the Ad script to get the data necessary for counting a viewer's clicks, but denies its requests to read cookies, passwords and other sensitive information. Our approach can also protect the Flash code through mediating the access of scripts and other SWF files to the call interfaces it exposes to its hosting page (Section 4.2).

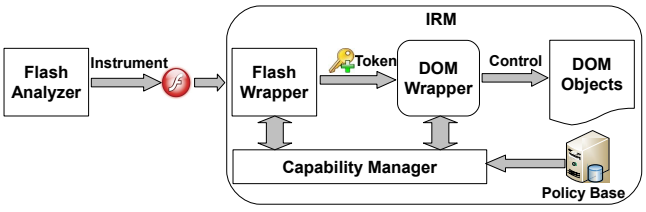


Figure 2: Overview.

To protect the IRM, our approach prohibits other scripts to wrap DOM functions (Section 4.3). This is enforced through regulating the methods (e.g., `__defineGetter__`, `__defineSetter__`) necessary for performing such an operation. The IRM also forbids unauthorized parties to read or write its code and the policy base, or access sensitive FIRM data, such as capability tokens.

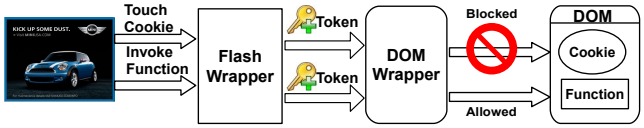


Figure 3: An example demonstrates the design of FIRM.

Though the DOM wrapper and the capability manager can be manually built into a web page by the web developer, an automatic tool is necessary for instrumenting SWF files with the Flash wrapper, as they are often developed by third parties and can be updated frequently. FIRM therefore provides a tool that automatically disassembles the binary code of a SWF file, identifies its access-related function calls (e.g., `getURL`) and internal functions exposed to other domains or JavaScript, and then wraps these functions with mediation code (Section 5).

4. INLINE MEDIATION

Inline Reference Monitor was proposed in [24] as a mechanism to mediate access to Operating System resources. It is built directly into an application's code to control program behaviors, and therefore does not need OS or hardware level supports. Similarly, the IRM used in our research is embedded in web contents, which avoids any changes to the client's browser. In this section, we elaborate our design and implementation of the IRM, which is composed of the capability manager, the Flash and DOM wrappers, and show how these components work together to mediate Flash behaviors and safeguard their own integrity and data confidentiality.

4.1 Capability

To mediate access to DOM and Flash functions, the IRM needs to know the privilege that the caller of these functions possesses. To this end, FIRM adopts a capability mechanism that requires each caller to produce a capability token to gain access. A capability [33] is a token that indicates a set of access rights a subject (e.g., Flash, scripts) has on objects (e.g., cookie, text item, functions). When the subject is about to access a protected object, this operation is checked against the capability: it is allowed to proceed only when the subject has the access right, as specified by a security policy associated with the capability. In FIRM, such capability tokens are maintained by the capability manager, which was implemented as a JavaScript program in our research. Following we elaborate how the mechanism works.

Capability management. Every subject with access policies specified by the web designer is assigned a capability token. An instrumented Flash application acquires its token from the capability manager when it is initialized. Specifically, the Flash exposes a callback function to the JavaScript, through `ExternalInterface.addCallback`. The function is called by the capability manager to parameterize the Flash with its token and the related security policies, and disabled by the IRM afterwards to prevent unauthorized invocations. This treatment avoids hard-coding tokens into these applications, a process that requires recompiling the applications each time their hosting web page is requested by a web client.

The capability token used in FIRM is a random string. It is designed to be sufficiently long ($\geq$ 10 bytes) to defeat a brute-force attack in which the adversary tries to use random guesses to produce a correct token. Each capabil-

ity token is associated with a set of security policies that specify a subject's access rights to different objects. The capability manager organizes those tokens and their policies into a *policy base*, which is stored in a local variable within a function called `Checker`. `Checker` encapsulates the policy base to mediate the access to its content. To retrieve policies, one has to call the function with a capability token. The IRM also hides its own capability in local variables and controls all the channels to read the code of `Checker` and its other functions. This technique is elaborated in Section 4.3. Another measure FIRM takes to prevent leaks of tokens to unauthorized parties is prohibiting a SWF file to share its capability with others. To this end, the mediation code our Flash analyzer injects into the SWF utilizes randomized, unpredictable names for the variables involved in capability-related operations to preclude any references to them from other part of the Flash code (see Section 5).

Table 1: Protected objects and properties

Type	Objects	Properties
DOM	Document	cookie, domain, lastModified referrer, title, URL
	Window	defaultStatus, status
	Location and Link	hash, host, hostname, href, search, port, protocol, pathname, toString
	History	current, next, previous, toString
	Navigator	appCodeName, appName, systemLanguage userAgent, userLanguage, platform
	Form	action
	Form Elements	checked, defaultChecked, defaultValue name, selectedIndex, toString, value
	Text	innerHTML, innerText
Flash	Functions	-

Security policies. Security policies are specified by the web designer for controlling subjects' access rights. A policy can be described by a 4-tuple $< s, o, a, c >$, where s, o and c denotes a subject, an object and the capability of the subject respectively, and a is the action s requests to perform on o. In FIRM, a subject is either a Flash application or the JavaScript code from a specific domain; an object describes DOM objects, or the functions or variables of JavaScript code and SWF files (see Table 1 for examples); the action element in the tuple can be "*read*", "*write*", "*execute*" or left blank to indicate denial of access.

Whenever a capability token appears with a request from a subject s' with an operation a' on the object o', the IRM searches the policy base to retrieve all the policies containing c'. The request is permitted if one of the policies contains s', o', a' and c', and denied otherwise. FIRM also includes a set of default policies specified by the web developer, which use a wildcard symbol '*' to match any subject, object, action or capability token. When the symbol is applied to c in the tuple, the policy is used on a subject that does not carry any capability. The default policies can be used to define a set of basic operations open to even untrusted Flash content or scripts, for example, a `read` on a nonsensitive text item, or avoid verbose specifications of the permissions for every subject/object pair. They are overruled by other policies once a conflict happens. For example, given two policies $< s, cookie, , c >$ and $< s, *, read, c >$, a Flash code s is denied the access to the cookie.

4.2 Mediation

The objective of inline mediation is to ensure that the web content under our protection can only be accessed by subjects with sufficient privileges, as indicated by their capability tokens. To this end, we designed and implemented a DOM wrapper and a Flash wrapper that work together to control both Flash/DOM interactions and inter-Flash interactions. We elaborate this approach below.

Flash and DOM wrappers. To mediate Flash code and its script's access to web contents, we need to control DOM and ActionScript functions used in such an access. This is achieved in our research through wrapping these functions with mediation code. Specifically, we developed two wrappers, the DOM wrapper that controls DOM functions and the operations of scripts, and the Flash wrapper that mediates the use of ActionScript functions.

```
Sample code for wrapping the getter of document.cookie
//1. get pointer to the old getter
var oldGetter = document.__lookupGetter__("cookie");
//2. define the new getter
function newGetter() {
   if(Checker(currentToken))
      return oldGetter();
   else
      throw "unauthorized access";
}
//3. replace the old getter with new getter
document.__defineGetter__("cookie", newGetter);
```

Figure 4: An example of redefining getter.

The DOM wrapper redefines the `get` and `set` methods of the DOM objects that need to be protected. Most DOM objects, such as document, window, forms, and the input box offer these methods for scripts to read or write their properties such as cookies, locations and others. In Mozilla Firefox 3.5, FIRM wraps these methods through `__defineGetter__` and `__defineSetter__`, two methods specified under `Object.prototype`. Other browsers, including IE8, Google Chrome, Safari and Opera, use different methods, as described in Table 2. The web server that implements FIRM can use scripts to identify the type of the browser a client is running (from `Navigator.appName`) before rendering a web page that wraps its methods. Figure 4 illustrates an example in which the `get` method of `document.cookie` is supplemented with the code that mediates the access to the property. Different from most properties, `document.location` and `window.location` do not have `get` and `set`. On the other hand, these two properties need protection because otherwise, untrusted scripts can modify them to redirect the browser to malicious websites. Our solution is to make use of the method `Object.watch` to monitor these properties: once the method detects that the properties are about to be changed by the party without a proper capability[1], the IRM simply aborts the redirection operation if the target is not permitted. Our approach also wraps DOM functions like `document.alert`, which pops up windows (e.g., an alert). These functions could be used in social engineering, and therefore need mediation. Other part of the hosting page that the DOM wrapper modifies includes the JavaScript code for accessing DOM objects or Flash resources (through the call interfaces exposed by SWF files). Such code is instrumented to add in the mechanism that checks the capability tokens of the party invoking it or attaches its token to every access request it makes.

The Flash wrapper controls ActionScript functions like `getURL`, `navigateToURL`, `ExternalInterface.call` and `fscommand`. These functions can be used to invoke the script

[1] `watch` can intercept the operations that modify the objects it is monitoring. An authorized party who wants to change the object needs to place its capability token to a "mark" variable, which is discussed later in this section.

Table 2: API variations in different browsers

Browser	Define Getter	Define Setter	Get Setter	Get Setter	Watch
Mozilla Firefox 3.5	__defineGetter__	__defineSetter__	__lookupGetter__	__lookupGetter__	watch
IE 8	defineProperty	defineProperty	getOwnPropertyDescriptor	getOwnPropertyDescriptor	onPropertyChange
Google Chrome 4	__defineGetter__	__defineSetter__	__lookupGetter__	__lookupGetter__	-
Safari 4	__defineGetter__	__defineSetter__	__lookupGetter__	__lookupGetter__	-
Opera 10	__defineGetter__	__defineSetter__	__lookupGetter__	__lookupGetter__	-

already in the hosting web page or inject new scripts. Individual SWF files stored in the hosting website are also instrumented with the code for acquiring capability tokens and their policies from the capability manager as soon as they are bootstrapped. This is achieved by exposing an ActionScript function to JavaScript, through which the capability manager parameterizes the SWF file. The mission of the Flash wrapper includes letting a Flash program use its capability to execute JavaScript code, and protecting its functions from being misused by scripts or other Flash applications.

Mediating access to DOM objects. A Flash application relies on JavaScript code to access DOM objects. To mediate the access, FIRM wraps all the ActionScript calls related to JavaScript, as discussed above. Whenever the Flash makes such a call, the Flash wrapper supplies the capability token of the Flash to the call, and the mediation code inside the JavaScript functions to be invoked calls `Checker` to look up the security policies regarding the token and makes access decision based upon the policies.

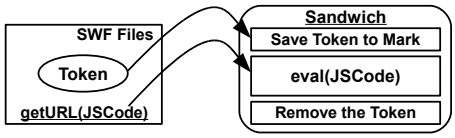


Figure 5: Sandwiching the injected script.

A challenging problem is how to let the JavaScript code injected by a Flash application run at the Flash's privilege. The IRM may not have access to such code until the runtime: for example, the code can be downloaded by the Flash from another site. Automatic analysis of JavaScript code is well known to be hard [27], which makes it difficult to instrument the code on the fly. We tackled this problem by leveraging a special feature of JavaScript: JavaScript code in a web page actually runs in a single thread, and as a result, its execution is sequential [42][2]. This feature allows us to develop a "marking" mechanism that labels the script code running on a Flash program's behalf. Specifically, the DOM wrapper maintains a "*mark*" variable, which is initialized to zero and later used to keep the capability token of the running script. After the Flash wrapper identifies a script injection operation in a Flash program, for example, from the prefix `javascript:` within the input content of `getURL`, it sets the script code as the input string to an `eval` command, and inserts one JavaScript command before the `eval` to set the mark to the Flash's capability and one after to zero out the mark. This transformation, which we call "sandwiching", is illustrated in Figure 5. When the script is running, the IRM refers to the mark for the script's privilege. Note that other scripts cannot read the mark before the sandwiched code runs to completion, due to the sequential execution of JavaScript [42]. On the other hand, the code cannot escalate its own privilege by changing the mark, as it does not know other capability tokens.

As stated in Section 2, a Flash application can redirect a visitor to a malicious site and install malware. To defend against this attack, FIRM mediates functions like `getURL`: if the input parameters of these functions are found to contain URLs (started with http, for example), they are used to check against a whitelist; only redirection to the URLs on the list are allowed.

Mediating access to Flash. A Flash application can choose to expose some of its functions (through Actionscript calls such as `ExternalInterface.addCallback`) to let JavaScript code access their resources. A problem here is that there is no restriction on who can call these functions. For example, a malicious Flash program can take advantage of these functions to gain access to another Flash that it is not allowed to access within the Flash player.

Our solution to this problem is instrumentation of the exposed functions. Mediation code, as part of the Flash wrapper, is inserted to the beginning of such functions after static analysis of Flash code. Once an exposed function is invoked, our code checks the capability token supplied by the caller, and decides whether to let the call go through according to the security policies tied to the token.

Inter-Flash access control. Adobe Flash player maintains a boundary between different Flash applications. Such a boundary, however, can be crossed if one Flash shares its functions to another Flash through a `LocalConnection` object. The problem here, again, comes from the "black-and-white" strategy adopted by the Flash player: a Flash program shares either *all* its functions or none at all. A serious consequence of this treatment is that untrusted Flash code can call the function of privileged Flash code to gain access to the resources it is not entitled to, once the latter inadvertently exposes its functions. Our solution to this problem, again, is based upon code instrumentation and call wrapping: for the Flash application that is found to build a `LocalConnection` with others, our analysis tool instruments all its functions with mediate code; the code checks the caller's capability token once a function is invoked, and aborts the call if the token does not carry a sufficient privilege.

A Flash application can load another Flash with `load` or `loadBytes` as a resource and then use `addChild` to make the latter its child Flash. When this happens, the child acquires the full access to the father's resources, including functions, variables and others, and is able to leak them out. FIRM mitigates such a threat by automatically reducing a Flash's privilege once it is found to have downloaded untrusted child Flash.

4.3 Protecting FIRM

Since an IRM works on the same layer as the subjects it controls, it is under the threats these subjects pose. Without proper protection, FIRM can be subjected to various

[2]The registered user events are triggered sequentially: they cannot be executed until the script stops running. Similarly, delayed execution with function `setTimeout` is also sequential.

attacks from malicious scripts [25] or Flash applications, including compromising the integrity of its code and policies, and stealing its capability tokens. In this section, we elaborate the measures our approach takes to mitigate these threats.

Integrity Protection. The obvious targets of attacks are the DOM and Flash wrappers. As Flash content is not able to alter its code in runtime (See Section 5), malicious Flash code cannot get rid of the Flash wrapper after it is instrumented. This feature automatically ensures the integrity of the Flash wrapper. Hence, our integrity protection is focused on the DOM wrapper.

As discussed before, the DOM wrapper mediates the `get` and `set` functions of important DOM objects. The adversary may try to replace the wrapper with his own functions. To eliminate this threat, our IRM has been designed to wrap these important objects before any other subject, and block any request without a proper privilege to change the getters and setters of the objects. This is achieved through mediating the methods `__defineGetter__` and `__defineSetter__`. To prevent malicious scripts from tampering with the wrappers for these methods, we employ `Object.watch`[3] (`onPropertyChange` in IE8) to monitor the operations on the methods: any change to their function pointers will be detected by `watch` and aborted by the IRM before it happens. The `watch` method itself is protected in the same way: it is watching itself and interrupts any attempts to replace it. The IRM also mediates all the methods of `prototype`, a property under `Object`, `Array` and `Function`. This is necessary for protecting the functions associated with these global variables, such as `toString`, which could also be modified by the adversary [19].

Prior research [36] discovered that a malicious script can delete all wrapped objects from the memory, which could lead to the restoration of the original, unwrapped objects. This threat, however, is limited to Firefox, and can be eliminated by setting constraints on the deletion operation, which is permitted under Standard ECMA-262 5 [12], the next generation JavaScript specification. FIRM also takes measures to mitigate the threat: once the IRM finds itself in Firefox and a Flash program is about to execute the scripts within the "sandwich" (See Section 4.2 and Figure 5), the Flash wrapper works with the DOM wrapper to calculate the hash values of the instrumented DOM methods and their function pointers and save them to the variables within the Flash wrapper. They also move all the valuable data, such as cookies, into the Flash wrapper. After the execution of injected scripts, the Flash wrapper verifies the integrity of these function pointers and methods. If no foul play is found, the valuable data is restored. Otherwise, it aborts its operation and warns the user through a pop-up window. The Flash variables used to save such data assume randomized names to protect them from being accessed by the original, uninstrumented Flash content. They are also beyond the reach of injected scripts, as they are located within the Flash.

Confidentiality Protection. The most sensitive FIRM data are capability tokens, which, once seized by unauthorized parties, can be used to escalate their privileges. During the operations of FIRM, these tokens are stored in the local variables of JavaScript and the variables of SWF files. Since these local variables cannot be referred by the script code outside their related functions, the only way an unauthorized party can access the capabilities is to read the code of these functions. This path is also blocked by the IRM, which is configured to allow none but itself to read its code, through mediating the `get` methods for the `innerHTML` and `innerText` properties under its `script` object, and the `toString`, `toSource` and `valueOf` methods under the prototype of `Function` object.

To protect the capability tokens stored in the instrumented Flash code, the names of the variables that accommodate these tokens are randomized, making them unpredictable to the adversary. Such an operation only needs to be performed once when instrumenting the code. As a result, a malicious Flash program is unable to access the capability tokens and other data stored in the Flash wrapper. Note that an instrumented Flash does not carry hard-coded capability token. Instead, it gets the token from the capability manager once it is bootstrapped by the Flash player.

As discussed in Section 4.2, the variables and functions of a Flash program are completely exposed to the child Flash it downloads. This lets the child inherit the father's capability, which can be risky in some circumstances. For example, a Flash-based video player could run an untrusted Flash Ad as its child. It is evidently undesired to grant the Ad the privilege of the video player. We solve this problem by instrumenting the ActionScript calls for downloading and creating a child Flash: once a child Flash is found to come from an untrusted domain, our mediation code automatically lowers down the father's privilege.

5. FLASH ANALYSIS AND INSTRUMENTATION

To embed the Flash wrapper into a Flash, we need to analyze its binary code and instrument the code with mediation mechanisms. As Flash contents are often submitted by the third party and in binary forms, manual analysis of such contents can be time consuming and even unrealistic for the Flash serving sites like Yahoo that receive a large number of uploads every day. In our research, we developed an automatic Flash analyzer to work on Flash code. Our analyzer can decompile a binary SWF file, identify the functions related to resource access and wrap them with FIRM instrumentation. In this section, we elaborate the design and implementation of this tool.

ActionScript. ActionScript is a scripting language based on ECMAScript [7], which is designed to control the behavior of Flash. Compared with JavaScript, ActionScript code is easier to analyze and instrument: 1) a Flash program is not allowed to modify its code during the runtime, which makes a malicious Flash impossible to get rid of our instrumentations; (2) Flash cannot parse and execute an input string as `eval` does, which avoids the complication in statically analyzing such a string; (3) a Flash program cannot access the code and data within another Flash without permission. These features allow us to perform a static analysis of Flash code to add mediation to the code.

Static analysis of Flash. The prototype we built first utilizes SWFScan [13], a free decompiler, to convert SWF binaries into the ActionScript code. Then, it identifies the program locations from the code where instrumentations need to be done. Specifically, our implementation looks for four types of ActionScript APIs: `getURL` and `navigateToURL` that

[3]According to ECMA-262 5 [12], any property of a JavaScript object has an attribute named `Configurable`. When set to `false`, assigning new value to the property will throw an exception, which achieves the same goal as `Object.watch`. Though the current versions of Google Chrome, Safari and Opera do not support the function, they will certainly move towards this standard.

allow a Flash to inject scripts into its hosting page, `ExternalInterface.call` and `fscommand` that enable the Flash to call a JavaScript function defined in the page, `ExternalInterface.addcallback` that lets Javascript call ActionScript functions, and `LocalConnection` that shares the functions of one Flash program with others.

To accurately locate these functions, we parsed the Flash code into a grammar tree. The parser implemented in our prototype was generated by ANTLR (or ANother Tool for Language Recognition) [3], a popular parser generator. We manually translated ActionScript grammar into the form accepted by ANTLR, which then converted it into an LL parser. From the grammar tree the parser creates, our analyzer can identify both the direct use of script-related APIs, for example, a call to `getURL()`, and the indirect use, for example, `var a = getURL; a();`.

Instrumentation. After relevant program locations are identified, our tool automatically instruments the code at these locations with mediation mechanisms. The mediation code allows the caller to supply its capability token for privilege checking, and invokes the original function if the caller is authorized. After instrumentation, the new Flash program is compiled into a SWF binary using the compiler of Adobe Flash CS [1].

Discussion. Though ActionScript is easier to analyze than JavaScript, it does include some language features that can be used to obfuscate its code. Particularly, ActionScript 2.0 allows the `_root` object to invoke a function through parsing an input string. For example, `_root['getURL']()` will be interpreted as `getURL()` during an execution. This technique, however, seems no longer supported by ActionScript 3.0. Moreover, some API functions like `asFunction` can be exploited to inject scripts [8]. To mitigate the threat, we can mediate their operations using the Flash wrapper. Note that all these language features are not frequently used by legitimate Flash code. On the other hand, a malicious Flash that uses them to obfuscate its code could end up decreasing its privileges, as uninstrumented calls cannot use any capability tokens. They are ensured by FIRM to have nothing but the lowest privilege.

6. EVALUATION

We evaluated our implementation of FIRM on real web applications and Flash contents. Our objective is to understand the effectiveness of our technique in mediating Flash activities, and the performance impacts it could bring to web services. In this section, we first explain our experiment settings (Section 6.1), and then elaborate on this experimental study and its outcomes (Section 6.2 and 6.3).

6.1 Experiment Settings

Here we describe the web applications, Flash and computation platforms used in our study.

Web applications. We utilized three extremely popular open source web-design systems in our study:

• *phpBB* is one of the mostly used open source forum solutions, with millions of installations worldwide [11]. It serves as a template, which one can customize, e.g., adding plugins, to build her own forum. The forums based upon phpBB allow users to post Flash contents through the tag `[Flash]` in *BBCode* [4].

• *WordPress* is a blog publishing application known as the largest self-hosted blogging tool, with millions of users worldwide [14]. Through the application, a blog author can publish Flash content, which is handled by a plugin called *Kimili Flash Embed* [9].

• *Drupal* is an open source content management system (CMS) that supports a variety of websites ranging from personal weblogs to large community-driven websites [6]. The system can be used to publish different types of web contents, including Flash.

Flash. Also used in our experiments were 9 real-world Flash applications, as illustrated in Table 3. Specifically, we utilized 3 vulnerable Flash advertisements, a malicious Flash game and a Flash player to understand whether our technique can effectively control the scripts they invoke. To study our protection of the call interfaces Flash exposes to scripts, an experiment was conducted to let malicious Flash code spawn scripts that attempted to access the functions another Flash exported to its hosting page. Inter-Flash access control was evaluated with another pair of Flash applications: one attempted to share only some of its functions, whereas the other tried to access other functions. All these Flash applications were instrumented with our analysis tool and executed within the aforementioned web services.

Computation platforms. All our experiments were conducted on a laptop with 3G memory and 2G Dual-core CPU. The laptop ran Windows Vista, with Apache 2.2.9/PHP 5.2.6 as web server and MySQL 5 as database server. Our experiments were conducted in Firefox 3.5 and IE 8.

6.2 Effectiveness

Installing FIRM. We modified these web applications to install the DOM wrapper and the policy manager. A phpBB-based forum could receive posts with Flash contents, for example, `[flash]a.swf[/flash]`, which will be activated in a viewer's browser[4]. In the absence of mediation, such a Flash can spawn scripts to compromise the integrity and confidentiality of the viewer's information assets. To embed our IRM into the web pages generated by the forum, we changed a PHP file `tpl/prosilver/viewtopic/body.html .php`, making the program inject the JavaScript code of the IRM into every web page it created. Similarly, a blog publishing system built upon WordPress can be used by the malicious blogger to spread Flash malware. In our experiment, we inserted code to `index.php` under the folder `wp-content/themes/classic` so as to embed our IRM scripts into the web content produced by the system. A website developed using Drupal can also inadvertently include malicious Flash contents, for example, an advertisement Flash that picks a web surfer's cookie. Like the other two applications, the system includes a PHP program `page.tpl.php` under `themes/garland` for dynamic page generation. This program was also modified in our research to build our IRM into its pages. We also ran our Flash analyzer to instrument the aforementioned Flash applications.

Experiment outcomes. To evaluate the effectiveness of FIRM, we first attacked the unprotected web applications through the Flash contents they hosted, and then made the same attempts on these applications when they were under the protection of FIRM. Following we elaborate our findings.

The mediation on Flash's access to DOM objects was studied using the five Flash applications described in Section 6.1 and Table 3. Without mediation, we found that the malicious Flash game could steal the cookie from the hosting page and the three Flash advertisements could be exploited to launch XSS attacks. Once the IRM was activated, all

[4]We changed a PHP file `bbCode.html` under `styles/prosilver/template` of phpBB3 to allow Flash executing JavaScript code.

Table 3: Effectiveness

Type	No	Flash	Operations	Result		
				phpBB	WordPress	Drupal
Flash to DOM	1	Puzzle Game	read cookie	Reject	Reject	Reject
	2	Adobe Demo	read cookie	Reject	Reject	Reject
	3	CNN Ad	change Location	Reject	Reject	Reject
	4	CNET Ad	read user account text	Reject	Reject	Reject
	5	Flow Player	O_1: read Location	O_1 Reject	O_1 Reject	Allow
			O_2: call script function	O_2 Allowed	O_2 Allow	
DOM to Flash	6	Color Widget	expose functions	-	-	-
	7	Invoker	call functions of Color Widget	Reject	Reject	Reject
Flash to Flash	8	Sender	call functions of Receiver	Reject	Reject	Reject
	9	Receiver	receive message	-	-	-

these attacks were found to be successfully deflected. Specifically, the instrumented phpBB and WordPress adopted the security policies that disallowed Flash contents to access any document objects. As a result, we observed that the JavaScript calls for accessing cookies, initiated by the Flash game and the Flash advertisements, were all blocked. A side effect was that the legitimate Flash, the Flow player, could not access the URL of the hosting page either. The policies specified for Drupal differentiated the legitimate Flash, Flow Player, from the other Flash applications: the player was allowed to access document objects, except the cookie, whereas the Flash games and advertisements were denied the access. We found that these policies were faithfully enforced by our implementation, which defeated all our attacks without interfering with the legitimate Flash's operations.

We employed two Flash applications to evaluate the protection of the call interfaces a Flash exposed to JavaScript. One of these applications share its functions with the hosting page, which could be accessed by the other Flash from a different domain through injecting script. After instrumenting the exposed functions with mediation code, we observed that the second Flash was no longer able to use the functions without legitimate capability tokens.

We also studied the situation when two Flash applications attempted to share resources between them. One Flash tried to let the other Flash call some but not all of its functions. In the absence of FIRM, this could only be done through establishing a `LocalConnection`, which unfortunately made all functions within the first Flash available to the other. After the IRM was installed, we could assign the second Flash a capability token that only granted it access to some of these functions. Such security policies were found to be successfully enforced by our prototype.

6.3 Performance

The performance of FIRM was evaluated in our research from three perspectives: (1) the performance impact of FIRM on page loading, (2) runtime overheads for mediating DOM and Flash operations and (3) the cost for performing static analysis on SWF files. We elaborate this study below.

Table 4: Performance of page loading

	phpBB3 (s)	WordPress (s)	Drupal (s)
No FIRM	3.927	1.66	3.555
FIRM	4.117	1.923	3.96
Overhead	4.80%	15.80%	11.40%

Bootstrapping FIRM could add further delay to a page loading process. To understand such a performance impact, we measured the page loading time of the three web applications described in Section 6.1 using a plugin (e.g., Firebug [35] in Firefox). The experiment was designed to compare the loading time of unprotected pages with that of instrumented pages. We collected the data from 10 independent tests under the settings when FIRM was installed and when it was not to compute averages, which are reported in Table 4. As we can see from the table, the overhead incurred by FIRM was reasonable: it was kept below 20% in the worst case (WordPress), and acceptable for other web applications.

Table 5: Mediation overheads

	Flash to DOM(ms)		DOM to Flash(ms)		Flash to Flash(ms)	
Tasks	Read cookie		Call func		Call func	
Browsers	IE	FF	IE	FF	IE	FF
No FIRM	0.641	0.799	6.52	4.92	0.675	0.67
FIRM	0.995	1.87	6.81	4.96	0.686	0.698
Overhead	55.20%	134.00%	4.40%	1%	1.60%	4%

We further studied the overheads incurred by instrumented operations, which include the delay caused by mediating the interactions between Flash and JavaScript, as well as those between different Flash applications. We collected data from 1000 independent tests of individual operations, with or without mediation. Averaged delays computed from such data are displayed in Table 5. From the table, we can observe that mediation was lightweight, incurring an overhead of 5%. A more significant delay appeared when the IRM was controlling the JavaScript code invoked by a Flash, which went up to 134%. A closer look at the overhead revealed that it was caused by `eval` the IRM employed to wrap the injected code (Section 4.2). Running JavaScript within the function turned out to be more time-consuming than a direct execution of the code in a hosting page. However, given the small execution time of the code, the delay introduced thereby was actually hard to notice in practice.

The overhead of analyzing and instrumenting Flash consists of the latencies incurred by decompiling binary code, analyzing, instrumenting and compiling the source code. We measured these latencies from the 9 Flash applications used in our study, each of which was run 10 times to get the average. The outcomes are presented in Table 6. The table shows that in most cases, the whole analysis took less than 10 seconds on the low-end laptop used in our experiment. Analyzing and instrumenting Flow player takes over one minute as it contained over 20,000 lines of code, while most Flash programs, particularly advertisements, are much smaller, typically below 1,000 lines.

Table 6: Perfomance of static analysis

Flash	Decompile (s)	Analysis (s)	Compile (s)	Total (s)
Puzzle Game	3.46	0.665	2.66	6.785
Adobe Demo	2.3	0.563	2.62	5.483
CNN Ad	4.9	0.723	2.8	8.423
CNET Ad	6.08	0.865	2.9	9.845
Flow player	14.6	3.582	49	67.182
Color Widget	3.6	0.848	3.98	8.428
Invoker	2.88	0.571	2.3	5.751
Sender	2.82	0.717	2.1	5.637
Receiver	2.5	0.631	2.74	5.871

7. DISCUSSION

FIRM is designed to be the first inline policy enforcement system that mediates Flash/DOM and Flash/Flash interactions. Also of great importance to Flash security, naturally, is well-designed security policies. The current design of FIRM can support simple policies, as described in Section 4.1. These policies seem to be sufficient for mitigating traditional threats such as XSS [29, 34, 18]. However, questions remain whether they offer enough protection against the new threats posed by malicious Flash, for example, seizure of the clipboard [2]. Further study is needed to understand this problem and improve FIRM to support more complicated policies, if necessary.

FIRM instruments the dynamic contents including Flash and JavaScript located at the websites that adopt our technique. For the Flash or scripts downloaded to the client's browser from other domains during the runtime, the control we could achieve is still coarse-grained: our current treatment just grants them the lowest privilege. A more desirable approach could be applying different policies to the dynamic contents from different domains. This could requires establishing certain trust relations between websites. Alternatively, our IRM could pass the scripts and Flash acquired during the runtime to its website (the one that offers the hosting page) for analysis and instrumentation. Study of these approaches is left to our future research.

As discussed in Section 4.3, a Flash can download and run another Flash as its child. The child Flash, which can be untrusted, inherits the privilege of its father. Our current solution is de-escalation of the father's privilege, which results in rather coarse-grained control. In the follow-up research, we plan to look into the possible approaches that can be used to mediate the child's activities without demoting the father.

The techniques we propose can be applied more generally: for example, they can be extended to mediate JavaScript code from different domains. On the other hand, our approach cannot protect a web service from a denial of service attack: for example, a malicious Flash or script can delete DOM objects to disrupt the normal operations of the service. Further research is needed to understand the feasibility of making IRM more resilient to the attack.

8. RELATED WORK

Inline reference monitor. The idea of moving a reference monitor into an application has been applied to protect binary executables [45, 16, 39] and Java applications [24, 20]. Compared with other access control mechanisms, an IRM is often more efficient and has more information about an application's internal states, but can also be more prone to the attacks that aim at its integrity and data confidentiality. Concurrently with this research and independently, Phung et. al. [36] proposed a JavaScript IRM that mediates accesses to sensitive DOM objects and properties. A problem with this approach is that all the scripts within a web page are granted the same privilege. In contrast, FIRM offers a fine-grained control of the scripts and Flash applications with different privileges, according to their capabilities. Another concurrent work from Meera et. al. [41] devised a Flash IRM to verify if certain functions violate pre-defined policies. To mitigate XSS attack, their framework sanitizes the input of the functions like `getURL`. Nevertheless, this approach is black-and-white which only allows or prohibits the whole script from input. Conversely, our framework can allow the legitimate script code while prohibits the malicious one.

Access control in web contents. The rapid development of new web services and applications, such as Mashup [10], makes the classic Same Origin Policy [38] increasingly insufficient for mediating dynamic web contents. New policy models and enforcement platforms, for example, MashupOS [44], OMash [22], xBook [40] and BFlow [46], are proposed to achieve finer-grained control of web activities, particularly those involving JavaScript. FIRM is designed to control Flash applications and the scripts they spawn, which has not been done before. Moreover, all these existing approaches require installing browser plug-ins. This raises the bar for their practical deployment. Our approach, however, does not need the web client to do anything: all the policies and enforcement mechanisms are completely embedded in the web pages delivered to the browser, and therefore can be deployed easily. Grier et. al. proposed a new browser named OP Browser [26] which embeds security policy in browser kernel to mediate the access from plugins like Adobe Flash player. However, their approach does not differentiate the security demands of different Flash contents and turns out to be too coarse-grained.

XSS defense. As a well-recognized threat to integrity and confidentiality of valuable web contents, XSS has received great attentions from security researchers. Prominent countermeasures include Beep [30], BrowserShield [37], Noxes [32], and BluePrint [43]. Different from the prior work, FIRM focuses on the XSS caused by vulnerable or malicious Flash applications. Controlling such a threat needs effective mediation of the interactions between Flash contents and JavaScript, which has not been explored by the prior research.

Instruction set randomization. FIRM protects its IRM through randomizing capability tokens, and the JavaScript and ActionScript variables that maintain those tokens and their related policies, which makes these critical resources out of the reach of malicious web contents. This idea has been inspired by previous research on Instruction Set Randomization (ISR) [31]. ISR was designed to defeat code-injection attack, through creating process-specific randomized instruction set. Recently, researchers move towards utilization of the technique to protect web applications. A prominent example is Noncespaces [28], which randomizes the namespace prefixes within a document to eliminate the scripts not created by the server. However, such a control can be coarse-grained: for example, the reference monitor either permits or denies execution of script code, but cannot decide what resources a running script can access.

9. CONCLUSION

Flash contents have been increasingly utilized for video

playing, advertising and other purposes. However, it is revealed [29] that Flash can be exploited by an adversary to launch various attacks, including XSS and XSRF. The intrinsic protection of Adobe Flash player is not sufficient in that it either denies a Flash's access to web resources or gives it unconstrained access. Patching such a security mechanism turns out to be nontrivial: Adobe Flash player is closed source, and deploying the patch on every browser cannot be accomplished easily. In this paper, we present FIRM, a novel solution that avoids these hurdles while still achieving effective mediation of Flash activities. FIRM builds an inline reference monitor into the web page hosting Flash contents. The IRM effectively mediates the interactions between Flash and DOM objects, and between different Flash applications, according to the capability token possessed by the Flash. Our approach protects the IRM through controlling DOM methods and randomizing the names of the variables that hold sensitive data, such as capability tokens. We implemented a prototype of FIRM and evaluated it on popular web applications, including phpBB, WordPress and Drupal, and 9 real-world Flash applications. Our study shows that the technique effectively protects user data, incurs small overheads and is convenient to deploy.

10. ACKNOWLEDGMENTS

We thank anonymous reviewers for their insightful comments. This work was supported in part by the NSF under Grant No.CNS-0716292 and CNS-1017782.

11. REFERENCES

[1] Adobe flash cs4. http://www.adobe.com/products/flash/.
[2] Adobe flash player clipboard security weakness. http://www.securityfocus.com/bid/31117.
[3] Antlr parser generator. http://www.antlr.org/.
[4] Bbcode. http://www.bbcode.org/.
[5] Cnn. http://http://www.cnn.com.
[6] drupal community pluminbing. http://drupal.org.
[7] Ecmascript. http://www.ecmascript.org.
[8] Flash url parameter attacks. http://code.google.com/p/doctype/wiki/ArticleFlashSecurityURL.
[9] Kimili flash embed. http://kimili.com/plugins/kml_flashembed/.
[10] Mashup dashboard - programmableweb. http://www.programmableweb.com/mashups.
[11] phpbb - creating communities worldwide. http://www.phpBB.com.
[12] Standard ecma-262. http://www.ecma-international.org/publications/standards/Ecma-262.htm.
[13] Swfscan. https://h30406.www3.hp.com/campaigns/2009/wwcampaign/1-5TUVE/index.php?key=swf.
[14] Wordpress - blog tool and publishing platform. http://wordpress.org.
[15] Yahoo! http://www.yahoo.com.
[16] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security*, pages 340–353, 2005.
[17] Adobe. Flash player security - controlling outbound url access. http://help.adobe.com/en_US/ActionScript/3.0_ProgrammingAS3/WS5b3ccc516d4fbf351e63e3d118a9b90204-7c9b.html, 2009.
[18] Y. Baror, A. Yogev, and A. Sharabani. Flash parameter injection. Technical report, IBM, As of September 2008.
[19] A. Barth, C. Jackson, and W. Li. Attacks on javascript mashup communication. In *Proceedings of Web 2.0 Security and Privacy 2009 (W2SP 2009)*, 2009.
[20] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with polymer. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 305–314, New York, NY, USA, 2005. ACM.
[21] S. Chenette. Malicious flash redirectors - security labs blog. http://securitylabs.websense.com/content/Blogs/3165.aspx, 2008.
[22] S. Crites, F. Hsu, and H. Chen. Omash: enabling secure web mashups via object abstractions. In *Proceedings of the 15th ACM conference on Computer and communications security table of contents*, pages 99–108. ACM New York, NY, USA, 2008.
[23] DP. Flash clicktag parameter xss. banks, e-shops, adobe and others vulnerable. http://xssed.org/news/98/Flash_clickTAG_parameter_XSS._Banks_e-shops_Adobe_and_others_vulnerable/, 2009.
[24] U. Erlingsson and F. B. Schneider. Irm enforcement of java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, 2000.
[25] Google. Attackvectors. http://code.google.com/p/google-caja/wiki/AttackVectors, 2010.
[26] C. Grier, S. Tang, and S. T. King. Secure web browsing with the op web browser. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 402–416, Washington, DC, USA, 2008. IEEE Computer Society.
[27] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the USENIX Security Symposium*, Montreal, Canada, August 2009.
[28] M. V. Gundy and H. Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *NDSS'09: Proceedings of the 16th Network and Distributed System Security Symposium*, 2009.
[29] P. Jagdale. Blinded by flash: Widespread security risks flash developers don't see. In *Black Hat DC 2009*. Hewlett-Packard, 2009.
[30] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 601–610, New York, NY, USA, 2007. ACM.
[31] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280, New York, NY, USA, 2003. ACM.
[32] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 330–337, New York, NY, USA, 2006. ACM.
[33] H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.
[34] S. D. Paola. Testing flash applications. In *6th OWASP AppSec Conference*, 2007.
[35] I. Parakey. Firebug - web development evolved. http://getfirebug.com/, 2009.
[36] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting javascript. In *ASIACCS '09: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 47–60, New York, NY, USA, 2009. ACM.
[37] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. In *Proc. OSDI*, 2006.
[38] J. Ruderman. The same origin policy. http://www.mozilla.org/projects/security/components/same-origin.html, 2008.
[39] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.
[40] K. Singh, S. Bhola, and W. Lee. xbook: Redesigning privacy control in social networking platforms. In *Proceedings of the USENIX Security Symposium*, Montreal, Canada, August 2009.
[41] M. Sridhar and K. W. Hamlen. Actionscript in-lined reference monitoring in prolog. In *Proceedings of the Twelfth Symposium on Practical Aspects of Declarative Languages (PADL)*, 2010.
[42] E. Stark, M. Hamburg, and D. Boneh. Symmetric cryptography in javascript. In *25th Annual Computer Security Applications Conference (ACSAC)*, 2009.
[43] M. Ter Louw and V. Venkatakrishnan. Blueprint: Precise browser-neutral prevention of cross-site scripting attacks. In *30th IEEE Symposium on Security and Privacy*, May 2009.
[44] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in mashupos. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP 2007)*, pages 1–16, 2007.
[45] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver, BC, Canada, August 2006.
[46] A. Yip, N. Narula, M. Krohn, and R. Morris. Privacy-preserving browser-side scripting with bflow. In *EuroSys'09*, 2009.

T-DRE: A Hardware Trusted Computing Base for Direct Recording Electronic Vote Machines

Roberto Gallo*
University of Campinas
Campinas, SP, Brazil
gallo@ic.unicamp.br
gallo@kryptus.com

Henrique Kawakami
KRYPTUS Cryptographic Engineering Ltd.
Campinas, SP, Brazil
kawakami@kryptus.com

Ricardo Dahab†
University of Campinas
Campinas, SP, Brazil
dahab@ic.unicamp.br

Rafael Azevedo
Tribunal Superior Eleitoral
Brasilia, DF, Brazil
rafael@tse.gov.br

Saulo Lima
Tribunal Superior Eleitoral
Brasilia, DF, Brazil
saulo@tse.gov.br

Guido Araujo‡
University of Campinas
Campinas, SP, Brazil
guido@ic.unicamp.br

ABSTRACT

We present a hardware trusted computing base (TCB) aimed at Direct Recording Voting Machines (T-DRE), with novel design features concerning vote privacy, device verifiability, signed-code execution and device resilience. Our proposal is largely compliant with the VVSG (Voluntary Voting System Guidelines), while also strengthening some of its reccomendations. To the best of our knowledge, T-DRE is the first architecture to employ multi-level, certification-based, hardware-enforced privileges to the running software. T-DRE also makes a solid case for the feasibility of strong security systems: it is the basis of 165,000 voting machines, set to be used in a large upcoming national election. In short, our contribution is a viable computational trusted base for both modern and classical voting protocols.

1. INTRODUCTION

Electronic voting systems (EVSs) are a very interesting subject, as they are comprised of system components which interact within an complex environment with boundary conditions of different nature, legal, cultural, logistical and financial. Several countries have adopted EVSs, tailoring them to meet their specificities.

The Brazilian voting system currently has over 135 million registered voters [2], with variable literacy degree. Thus, electronic voting is a very simple procedure, which consists of typing candidates' numbers on a reduced keyboard, guided by simple instructions on a small screen. Brazil adopted Direct Recording Electronic voting machines (DREs from now on) in 1996. In 2009 a decision was made to replace part of the aging hardware base with a newly designed version, while maintaining backward compatibility.

Voting Systems Fundamental Goals

In spite of local constraints, EVSs share six common, fundamental, goals (Sastry [24]):

Goal 1. One voter/one vote. The cast ballots should exactly represent the votes cast by legitimate voters. Malicious parties should not be able to add, duplicate, or delete ballots.

Goal 2. Cast-as-intended. Voters should be able to reliably and easily cast the ballots that they intend to cast.

Goal 3. Counted-as-cast. The final tally should be an accurate count of the ballots that have been cast.

Goal 4. Verifiability. It should be possible for participants in the voting process to prove that the voting system obeys certain properties.

Goal 5. Privacy. Ballots and certain events during the voting process should remain secret.

Goal 6. Coercion resistance. A voter should not be able to prove how she voted, to a third party not present in the voting booth.

These goals are related (e.g. a voting system that does not satisfy goal 5 will hardly satisfy goal 6) and potentially conflicting (e.g. it is not trivial to build a voting system that is totally verifiable while preserving voters' privacy). Third-party end-to-end verifiability has been a recurrent subject [20]. Usually, verifiability is linked to the concept of (statistical) confidence level. Different cultures, and thus electoral laws, have different thresholds for the level of confidence they consider adequate for the electoral process.

*Partially funded by KRYPTUS and SERASA Experian research grants

†Partially funded by FAPESP (2007/56052-8), CNPq (309491/2008-8), and SERASA Experian research grants

‡Partially funded by FAPESP (2010/14492-4) and CNPq (305371/2009-6) research grant

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

Software independence is not enough. Different voting protocols [3, 17, 5] have been proposed to meet the above goals, with variable degrees of success and effectiveness. Unfortunately, most of them can be defeated by compromised software or hardware running in the underlying computing base. In order to mitigate such threats, software-independent systems were proposed by Rivest and Wack [21]: *A voting system is* software-independent (SI) *if an undetected change or error in its software cannot cause an undetectable change or error in an election outcome.* However strong, this concept ensures most of the above requirements but not all.

For instance, coercion resistance and vote privacy are especially susceptible to attacks based on tampered hardware and software, as vote input devices themselves can leak information [12, 22, 24]. Hardware protection and verification is thus an essential aspect, regardless of whether SI systems are employed or not. While some effort has been done towards the specification of hardware functionalities in order to provide sufficient device accreditation and tamper resistance [19, 8, 24], there is much room for improvement on the path to feasible implementations. Here we follow that path, presenting a hardware trusted computing base (TCB) for direct recording electronic voting architecture, T-DRE in short, suitable for a variety of existing voting protocols and systems.

Summary of our Contributions

Our contributions are present both in the novelty of the T-DRE components and in their composition. Namely, we propose a trusted hardware architecture that extensively employs signed code execution with hardware-enforced access control to peripherals in order to prevent a number of attacks. Further advancements include human-computable device integrity verification mechanisms, strong accountability, and improved signed-code execution assurance, all supported by a certification hierarchy which takes advantage of the proposed hardware.

The T-DRE architecture described herein was adopted by the Brazilian National Election Authority (Tribunal Superior Eleitoral - TSE). In order to fully validate the specification, we first implemented a prototype evaluation platform. Subsequently, the specification was realized by a vendor under TSE's control, using another hardware platform, and taking into account additional costs and stringent field, legal, and resilience restrictions, while maintaining backward compatibility with the deployed base. This endeavor, which resulted in 165,000 produced units, further supports our claims on the feasibility of the architecture.

Our proposal is not an airtight solution to electronic voting; we discuss its limitations in Section 5. However, we do claim that it provides a layer of security to SI and non-SI systems alike, whose strength is degrees above that of voting systems currently deployed around the world, by making it extremely difficult and costly for a fraud attempt to go undetected. Also, although we target centralized elections, in Section 4.2 we discuss how T-DRE can be naturally extended to decentralized environments such as in the USA.

This paper is organized as follows: Section 2 gives practical goals and boundary conditions of voting systems; Section 3 discusses related work; Section 4 details our proposal; Section 5 reports implementation efforts; Section 6 concludes, with ideas for future work.

2. VOTING SYSTEMS PRACTICAL GOALS AND BOUNDARY CONDITIONS

Attaining the fundamental goals is subject to practical boundary conditions, especially in large elections. Three important constraints are:

Availability. Voting systems must be available during the critical periods (election day, tallying, etc.) and resist denial of service attempts. DRE machines must resist tampering;

Credibility. An aspect of utmost importance, it is at the basis of fair representativity. Accordingly, implementations of voting systems should minimize the chance of operational errors and resist tampering. Here, again, DRE hardware security and verifiability plays an important role;

Resource Rationalization. The practical realization of voting systems should take into account various cost-related variables, such as auditing and hardware cost and maintenance. When security is considered, a clear budget trade-off exists between built-in security mechanisms and the security procedures employed by the Electoral Authority (EA). While the first is typically a one-time expenditure which is multiplied by the number of DRE machines, the second is recurrent, flexible, and proportional to the number of polls. The security targets for DRE machines must take this into account.

Security Targets

The specification of security targets should make provisions for many different variables (Common Criteria [27]). In face of the current Brazilian Electoral Laws, the following variables demand special attention:

Window of opportunity. Our implementation should take into account that attacks on DRE machines can occur at any time, but more easily in the interstices between elections. Pre-election time is the most vulnerable due to transportation of DRE machines across huge distances.

Surface and scope of attacks. Voting machines are subject to different levels of adversarial exposure between procedural checkpoints established by the EA: during election interstice, an adversary can have physical access to the DREs; in the pre-election (setup) phase, adversaries may have media (logical) access to the DREs; at election day, adversaries typically have only operational access to DREs, as all non-HID I/O are sealed and the machines operate offline. Our security target take these conditions into account. It provides tampering resistance and tampering evidence on the Critical Security Parameters (CSP) such as keys and key counters, with a physical security target of FIPS 140-2 level 3 [18] (passive resistance). Moreover, a successful attack must have limited scope - breaking one DRE should not increase the chances of an adversary of breaking another.

Level of adversarial expertise. Attacks on a DRE, especially those which adulterate or recover key material or CSPs, must demand multiple experts, considerable

time (impossible to execute during election day) and removal to a laboratory with special equipment.

Audit control points, mechanisms and equipments. Audit points shall be precise, clear and accessible. There should be an audit point aggregator that simply expresses the DRE's state (fully operational, in error, in service). The interpretation of this audit point should not require additional equipment nor complex procedures, being accessible to all parties involved in the electoral process: voter, electoral authority, poll worker, and party advocates.

3. RELATED WORK

In this section we discuss related work regarding T-DRE's features.

3.1 Signed Code Execution

Signed code execution [4, 1] is an important tool in voting systems [23, 28]. Many security issues faced by EVSs can be directly mitigated by the proper use of signed code execution. Benefits include:

- ensuring that only official voting software is executed in DREs, enhancing resilience against deliberate adulteration and operational errors which may violate EVS fundamental goals such as vote secrecy and coercion resistance;
- tracing and accountability of incidents, enabling security through legal means;
- simple verification of binaries' integrity in pre, intra and post-election phases, which facilitates auditing by parties, voters, and the Electoral Authority.

Hardware-based signed code execution can be achieved by various means, the *de facto* standard being the Trusted Computing Group (TCG, now ISO/IEC 11889) Personal Computer Trusted Platform Module (TPM) [11], a companion chip to the main system CPU, usually connected via LPC bus. The TPM has functional characteristics similar to a smart card. In cryptographic terms, the TPM performs several operations: key generation, storage and use of cryptographic keys, protected by a key that represents the system's root of trust. Moreover, unlike typical smart cards, the TPM has mechanisms for software attestation, which allows certain running application parameters to be anonymously verified and certified as not tampered. The module is recommended by the VVSG ([28], Section 5.5.1) for protection of the DRE software stack.

One of the drawbacks of PC TPM modules is that they work passively, in hardware terms, with respect to the main system CPU. TPMs, by design, can be completely bypassed by the system's boot sequence if the BIOS (especifically, the "Core Root of Trust for Measurement", CRTM) is tampered with, and thus "deceived" when used in application verification tests. Extensions to the TPM as the TEM from Costan et al [6], being also passive with respect to the CPU, represent no improvement in this regard.

To overcome this master-slave problem, one can consider the sole use of secure processors as the main component of a TCB aimed at DREs. However, even state-of-the-art processors with security features, such as AEGIS [26], USIP-PRO [13] and Cell [25], suffer from impeditive shortcomings. While the AEGIS specification is completely open, to the best of our knowledge there are no commercially available realizations of it. The USIP-PRO, in turn, has limited processing power, its architecture is proprietary and the vendor makes no assertions regarding memory protection against data modification. Finally, the Cell processor is proprietary, not allowing full access to hardware features from independent software vendors, thus adding undesirable obscurity to the design.

3.2 Key Management and Certification

Entertainment platforms have guided the industry regarding the execution of signed code for DRM purposes. Microsoft's Xbox [10] and Sony Playstation 3 execute only code signed by keys directly under vendors' root CAs. With the Cell Processor [25], Sony advances further: unsigned code running on PS3 has limited access to the device's peripherals, notably the GPU. Only signed code has full access to hardware features. The VVSG (Section 5.5.1) forbids non-signed code from running on DRE hardware, similarly to console platforms. The VVSG also recommends a TPM-like component for controlling software execution.

In addition to certifying (signing) the voting machine software stack, cryptographic key material is extensively used in many voting systems [28, 23, 3, 16, 17] for other reasons, from voting, to producing closeout records, audit log signature and verification, to encryption/decryption of votes and other sensitive material.

Although key management and storage could be handled in software by the DRE, cryptographic tamper-resistant hardware is preferred. The VVSG recommends the existence of a hardware tamper-proof signature module (SM) in DREs, whose primary function is to manage the life cycle of two asymmetric key pairs: i) the Election Signature Key (ESK), a unique per-election/per-device key used to sign votes and closeout records; and b) a per-device DRE Signature Key (DSK), which identifies the device and is used to produce certificates for the ESK. The usage of DSK and ESK is strictly controlled by the SM by means of two counters: CountESK and CountDSK. CountDSK counts the number of generated ESK certificates ever signed by DSK. CountESK counts the number of ESK usages. When the closeout record is produced, ESK is erased by the MSM and both counters are included in the resulting record.

3.3 DRE System Verification

Easy auditing is a paramount requirement for voting systems as it is central to the establishment of trust on the DREs' integrity and correct operation. The concerns with integrity verification of the entire DRE system stack (hardware, firmware, and software) are not new. Although auxiliary devices (software or hardware) can be used, ideally solutions should provide effective user-computable verification mechanisms of the DRE integrity, so that less, not more, hardware and software components are used to verify the main system. In this sense, device integrity verification itself should be also software-independent.

Sastry [24] describes a handful of desired DRE verifying properties, mainly aiming at software insulation, by constructing a proof-of-concept DRE with multiple (seven) processors. Gennaro et al [9] establish a condition for tamper-proofness of general hardware and give some clues on how to check device integrity by means of cryptographic chal-

lenges. Öksüzoglu and Wallach [19] present, in VoteBox Nano, an elegant human-verifiable software and firmware (FPGA bitstreams) checking mechanism based on random "session identifiers", which change every time the DRE is rebooted. Gallo et al [8] generalize Gennaro et al's conditions, prototyping a human-readable, cryptographically-strong system verification method called Time-Base One-Time Verification (TOTV), which allows for multiple device verification in a trust amplifying fashion, making humans part of the verification protocol. Although both [19, 8] can be used by poll workers and party advocates to assert DRE integrity, they are not practical for large-scale verification by voters, as they require comparison of multiple digit verification numbers, a hindrance when illiterate voters are considered.

4. OUR PROPOSALS

4.1 The T-DRE Architecture and the Master Security Module

The T-DRE architecture was devised to meet security and availability requirements, as well as cost restrictions. Some key requirements are:

- **(R1)** Run solely signed code, even if the opponent has operational access to the DRE media.
- **(R2)** Enforce the verification of the entire software stack, from the BIOS to the voting application, establishing an effective software trust chain;
- **(R3)** Allow the system state (integrity) to be widely attested by any user. Voters, party advocates and the electoral authority (EA) should be able to verify the integrity of the DRE without additional electronic devices;
- **(R4)** Resist physical and logical attacks, preventing unauthorized access to key material and application tampering;
- **(R5)** Contain only fully auditable components, enabling thorough system verification by the EA and the society;
- **(R6)** Allow the use of low cost, widely available hardware components, with reasonable computing power and fully open source development chain;
- **(R7)** Allow maintenance of the DRE machine and upgrade of its cryptographic mechanisms during its long expected lifetime (10 years);
- **(R8)** Enable and ease software and firmware development cycle, including field testing and simulations; allow faithful simulations which are clearly verifiable as such, which includes the production of non-valid results only.

In order to achieve these objectives, we based our proposal on the fundamentals of secure hardware presented by Gennaro et al [9] and Gallo et al [8]. The latter introduces the concept of *cryptographic identity*, which states conditions for the establishment and verification of a root of trust for general secure hardware. Both suggest the use of their verification schemes in DREs. Here we go further, presenting a DRE system architecture which also brings new control mechanisms and a new verification method (Section 4.3).

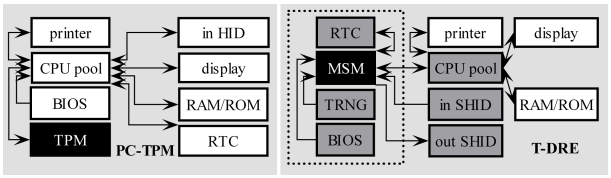


Figure 1: PC-TPM architecture (left) and the T-DRE architecture. The T-DRE components surrounded by the dotted box are under physical protection; BIOS physical protection is optional. Dark-gray components are under MSM direct control.

Our architecture is depicted in Figure 1, along with a classical PC-TPM system. In both, the CPU pool (one or more main processors) is the main processing unit, which runs the voting application (and software stack). In the PC-TPM design, the CPU pool is the bus master of all peripherals, including the TPM chip, which can be completely bypassed by tampered software at boot time. There is no way for the TPM to prevent CPU access to peripherals, nor to inform users that non-signed code is running.

The T-DRE Architecture, in contrast, is fundamentally different from the PC-TPM: the security is based on the proposed Master Security Module (MSM), which concentrates the DRE's cryptographic mechanisms and controls system peripherals (encrypted voter keypad, poll worker terminal, status lights), BIOS, and CPU pool. This centralization allows for a multi-level certification-based peripherals' access policy which can be enforced on the software running on the CPU pool. This is further explained in Section 4.2. The MSM control over the human interface devices (HID) also plays crucial role in our solution. Its implications are explored in Section 4.3. The MSM is also a CID-enabled device, i.e. a device whose root of trust, represented by a cryptographic key, is bound to the device's physical integrity: crossing the cryptographic boundary is highly likely to cause the device's root key destruction (and thus its identity), preventing the production of valid closeout voting records.

The T-DRE Software Verification, in contrast to PC-TPM, allows for full software stack verification, including BIOS. Prior to the CPU boot, after the DRE hardware power-up, the MSM checks the authenticity (and possibly decrypts) the BIOS contents; only if a valid (signed) BIOS is found, the CPU pool is able to boot. Now the CPU runs signed code from the very beginning of the boot sequence and is able to use the MSM to check the remaining of the software stack (bootloader, O.S., voting applications, scripts, configuration data). The differences between the T-DRE and the PC-TPM boot processes are illustrated in Figure 2. It goes beyond VVSG's required signed code verifier hardware module (VVSG, Section 5.5.1).

Both the T-DRE peripheral architecture and the software verification mechanisms are novel to DREs. Moreover, the MSM also acts as a VVSG Signature Module (VVSG Section 5.1.2). In spite of these advancements, our architecture can be implemented with off-the-shelf electronic components, enabling secure, fully auditable systems and low cost realizations. In Section 5 we describe a prototype using only commodity, general purpose components.

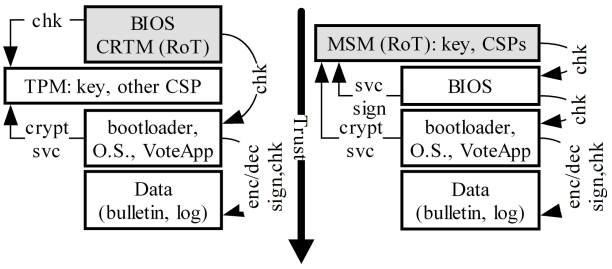


Figure 2: Verification chain for code execution, PC TPM and our proposed MSM

4.2 Hardware-Reinforced Certification-Based Privileges

Satisfying Section 1 goals (in special privacy) and Section 4.1 requisites (in special R3, 5, 7, and 8) requires strict control over the DRE software. Only official (highly audited) voting software must be able to produce valid closeout records. Maintenance (loosely audited) software must be prevented from accessing the DRE's key material (thus preventing production of valid closeout records) and from running an apparently valid, but otherwise fake poll (thus breaking privacy). Also, voting software being developed must be able to exercise all DRE features without being able to produce valid tallies or deceiving voters.

To attain the desired software control, we combined the MSM's control over the DRE's peripherals and the running software stack, with a custom key hierarchy based on Public Key Infrastructure (PKI) technology (with established procedures and audit controls), thus reducing required audit points. Our proposal centers the confidence of the electoral system on the EA root certification authority (EA-rootCA), which is audited (cryptographically) by the parties and the society. Figure 3 illustrates the PKI architecture with its three intermediate CAs, VoteCA, DevelCA, and ServiceCA, each with distinct purposes and privileges. In common, these CAs are responsible for": a) managing the DSK certificate life cycle; b) signing the DRE's software stack; and c) decrypting any messages coming from the DRE, when the voting protocol so demands. Software signed under each certification branch has different execution privileges and access to different key materials. Each DRE has three DSK certificates (and key pairs), one for each tree branch. *All DRE certificates (and corresponding keys) are stored within the MSM, which controls both the key usage and the signed code execution privileges.*

Vote CA Branch: Binaries signed under this branch have total control over the DRE hardware and are used in the actual election days - they have access to the official voting key material (DSK_{vote}, ESK_{vote}), producing valid election closeout records, controlling the voter's keypad use, the poll worker's keypad use, and the access to the Secure Output HID (Section 4.3). The MSM is responsible for enforcing the privileges of the signed code over the DRE hardware, without any software interference.

Development CA Branch enables the necessary functions for development and election simulation activities , granting restricted access to peripherals and keys: i) the MSM produces signatures only with DSK_{devel}, ESK_{devel}, $Other_{devel}$ keys; and ii) the signed code has no access to the

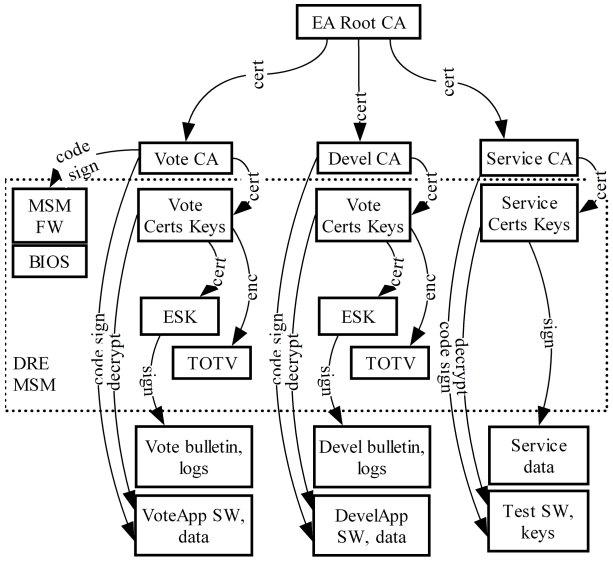


Figure 3: Certification hierarchy, code and data, and key usage

CA Privilegies	VoteCA	DevelCA	ServiceCA
Key Material	$(DSK)_{vote}$ $(ESK)_{vote}$ $(Others)_{vote}$	$(DSK)_{devel}$ $(ESK)_{devel}$ $(Others)_{devel}$	$DSK_{service}$
Input HID access	Full	Full	Restricted
Output HID access	Full	Full	Only test results
Security API (Secure HID)	Full	Restricted	None

Table 1: Signed code execution privileges for our DRE proposal; MSM enforcement

secure output HID which signals valid polls. This prevents in-development code from being used to deceive voters, and easily distinguishes valid signatures on real closeout records from those produced under simulation.

Service CA Branch enables DRE maintenance (memory, battery, peripherals testing and systems components replacement). Servicing operations are highly distributed, thus hard to audit. Under ServiceCA, signed code is not allowed access to keypads nor the secure output HID nor any ESK key material. The allowed operations are: a) re-pairing the input/output of cryptographic devices, and b) signatures of maintenance logs. Table 1 summarizes the privileges enforced by the MSM in each certification branch.

Other Considerations: Although our proposal targets centralized elections, it can be naturally extended to decentralized scenarios, as those in the USA, by adding Local Electoral Authorities CAs (as additional intermediate CAs) to the tree of Figure 3. Then, each local authority would maintain three CAs (VoteCA$_{local}$, DevelCA$_{local}$, ServiceCA$_{local}$). This allows a great deal of independence and flexibility, where local authorities can produce and run their own software without depending on the national authority. Furthermore, DREs can be easily shared by local

authorities.

4.3 T-DRE Verification: Secure Human Interface - S-HI

Integrity verification schemes provide variable confidence level in their output. As a rule, the better the scheme the more intrusive an adversary has to be in order to fake a result. From less to more intrusive we list: software modification (SWM), hardware modification (HWM), and key extraction from hardware (KXT). Human verification is especially hard to attain if tampering with the communication channel between the user and the system under verification is a possibility. *We call a human interface secure (S-HI) up to a class of intervention (S-HI-SWM, S-HI-HWM, S-HI-KX) if it does not produce false results even when it is subject to tampering of that class.*

The VoteBox Nano random number display (along with its verification scheme) is S-HI-SWM, i.e., it resists logical (bitstream) attacks, but not S-HI-HWM. In T-DRE we provide users with two interfaces: one S-HI-SWM and one S-HI-HWM. For the S-HI-SWM interface, we employ the MSM (hardware-)controlled 'out SHID' (Figure 1) as a four-state LED which indicates VoteCA, DevelCA, ServiceCA, and non/corrupted signed code. This is a clear improvement over VoteBox nano, as we attain the same security level with a much simpler user verification scheme.

For the S-HI-HWM interface, we employ a modified version of TOTV [8] that does not require the high-stability secure real-time clock (HSSRTC) of Gallo et al's solution. The TOTV protocol is similar to the Time-Base One-Time password (TOTP) described in [15]; TOTP derives, from time to time, an n-digit sequence from a secret key known to the *verified* device and possibly to the *verifier*. It is defined as $TOTP = HOTP(K, T)$ where T represents the number of time steps between the initial counter time T_0 and the current Unix time. K is a key, and $HTOP$ is the HMAC-based One-Time Password Algorithm defined (RFC 4226 [14]) as $HOTP(K, C) = Trunc(HMAC - SHA - 1(K, C))$. The TOTV proposal binds the secret derivation key K to the device's cryptographic identity (CID), so that any attempt to tamper with the device, by construction, should destroy the CID and thus cease the TOTV sequence creation. In our architecture, we maintain two TOTV keys (K_{vote}, K_{devel}) protected by DSK_{vote} and DSK_{devel} keys.

In order to check the integrity of a specific DRE, a user has to access a TOTV sequence produced by the electoral authority. In other to avoid replay attacks, this access must be either i) confidential and prior to the DRE display of the TOTV, or ii) real-time, on-demand, and signed.

In our proposal, we use the same construction as the TOTV, but instead of having a single T representing the number of time steps since Unix epoch, we use two T variables (T_{vote}, T_{devel}). These represent the time steps accumulated during every DRE usage when running in voting mode and development mode, respectively. The time counters necessary for this are made persistent and are protected by the MSM from stalls or decrement. In order to avoid other types of replay attacks, and after signed closeout records are produced by the DRE, it stalls the counter and includes it in the certificate, pausing the timing increments. In the next DRE usage (possibly on the next election), the electoral authority sends the poll workers $TOTV = HTOP(K, T)$, with $T = max(T_{closeout}, T_{user-access})$, which allows for DRE boot-up and counter resumption. The modification from the original TOTV proposal is motivated by the cost of a high stability secure real time clock. The usage of our proposals is further illustrated in Section 5.

5. T-DRE IMPLEMENTATION & RESULTS

The practical realization of our proposals was done in two phases, a prototyping and a mass production phase. In the first, the theoretical, technological, and procedural solutions were tested and validated. In the second, any necessary modifications were implemented.

5.1 Hardware and Firmware Implementation

Prototype Due to the large number of DREs to be produced (165,000), our proposals were thoroughly tested in a prototype prior to the delivery of final specifications to the chosen vendor for mass production. In the prototype (composed by two connected boards: B1 and B2), we instantiated all of the T-DRE main peripherals (Figure 1, namely: MSM, BIOS memory, encrypted voter keyboard (in SHID), output device (serial display), secure output (out SHID), main CPU, among others. The B2 board is a commercial embedded PC, with an AMD Geode LX800 CPU, with 256MB RAM. The B1 is a custom board specifically built for the prototype. It hosts the MSM and other devices, and connects the security module to the bottom board by means of an ISP connection (to BIOS delivery) and a USB connection (for other, cryptographic, services).

Considerable effort was spent on the correct choice of the micro-controller (uC) employed for the MSM as it must conform to many requirements: a) have internal code and data memory (both persistent and volatile); b) the entire memory must be lockable (no read/write access); c) memories must be large enough to handle cryptographic mechanisms (RSA, ECDH, ECDSA, SHA-2, homomorphic DH) and store keys and certificates; and d) reasonable performance, in order to handle quick BIOS verification and cryptographic services.

In our prototype, the MSM was implemented using a NXP LCP2000 (ARM) family uC which meets these requirements: a) up to 1MB internal FLASH memory with code read protection, b) up to 40KB RAM, enough for the implementation of asymmetric algorithms; c) 72MHz, 32-bit core, with 64 DMIPS performance. The voter input device (cryptographic, tamper-resistant physical keyboard) was simulated using a MSP430 uC, connected to the main uC by an SPI bus. The output secure HID is composed by three light emitting diodes (LEDs) which are directly connected to the MSM. In order to provide an onboard source of entropy, we implemented two random number generators using avalanche-effect semiconductor noise.

For the asymmetric algorithms on the MSM and the cryptographic keyboard we used the RELIC library [7]. For our prototype, the implementation of the required MSM functionalities, including DSK and ESK handling, binary code verification, CSR exportation, secure firmware update and cryptographic keyboard handling required about 180Kbyte FLASH (code) memory and 24Kbyte RAM. Employed functions were: signing and verification, asymmetric encryption/decryption (RSA-2048 PKCS#1); hash (FIPS 180-3 SHA-512); block ciphers (FIPS 197 AES 256).

A prototype software stack was also implemented. The bottom board BIOS was modified so that it uses the MSM slave interface to check the bootloader's authenticity. The

bootloader was also modified (from GRUB) to test the boot image, rather than files, using the MSM.

5.1.1 Attacks and Countermeasures

T-DRE, as PC-TPM, has no effective runtime (after boot) countermeasures against defective software nor buffer overflow attacks (data execution). While the first problem can be traced (and later dealt with) due to the sole use of signed code, the second demands more attention. In Brazil DREs have no data links, so buffer overflow attacks from voters or poll workers keypad is highly unlikely. For further protection, one may consider the "reboot prior to each vote" approach.

Hardware systems are subject to many implementation attacks, in special side-channel analysis (SCA) [12]. SCA use information leaked through side-channels from real systems. More information can be found in [12] and [22]. SCA-aware cryptographic hardware usually resists, to a certain extent, side-channel attacks. However, they typically suffer from lack of transparency on the employed security mechanisms (see Section 3). As we privilege transparency over off-the-shelf solutions, our solution uses a standard uC and added FIPS 140-2 level 3 equivalent physical protection and SCA counter measures:

- The entire top board was immersed in tamper-resistant and -evidencing resin;
- In order to weaken power attacks (SPA, DPA, CPA), we adopted two countermeasures: a) we used decoupling elements in all external communication paths; and b) we filtered and stabilized the power input to prevent energy consumption variation;
- Timing attacks are weakened by using constant-time cryptographic operations.

5.1.2 Mass Production Versions

After validation, our architecture was realized in a mass production version, and is set to be used on the 2010 Brazilian national election, with more than 165,000 DREs. This version differs from our prototype in some implementation decisions and functions: a) there is a single board containing all the components required in our architecture; b) the CPU pool was implemented as a single x86 processor; c) the MSM master interface was replaced by an assistive (supervisor) interface; if the MSM perceives any BIOS change, it resets the CPU pool (the main drawback being that BIOS cannot be encrypted). A second mass production version is expected to be manufactured in the fourth quarter of 2010, with more than 200,000 DREs. These will present further side-channel countermeasures and incorporate improvements deemed necessary.

5.2 Usage Procedures

5.2.1 Pre-Election, Election, and Post-Election Procedures

Since valid (non-tampered) voting machines run only code signed by the electoral authority, it is easy for a verifier to check whether the voting application is correct and that the voting machines have not been tampered with:

- In the **pre-election** phase, a human verifier must: a) Check for any physical tamper evidences on the DRE; if any are found, stop and report; b) switch on the DRE and enter the "resume TOTV" provided by the electoral authority (Section 4.3); if the DRE fails to continue the boot process, stop (either it is not the correct DRE or the device has been tampered with); c) check for the next TOTV to be shown by the DRE; if it is not the expected one, stop (the DRE has been tampered with); d) perform other verification procedures (e.g. audit procedures).
- On **election day**, human verifiers can, at any time: a) check for software stack integrity, by simply checking a DRE's status S-HID (indicative LED); if the S-HID does not present a valid status, the use of that DRE must be prevented (either it has been tampered with or it is not running the correct voting software stack); b) from time-to-time, electoral judges and voters can check for device integrity by comparing the TOTV produced by the DRE with those from the electoral authority; if any comparison fails, stop that DRE's use (it has been tampered with).
- In the **post-election** phase, a human verifier must check whether the final TOTV present in the closeout record is valid; if not, the device has been tampered with and the produced closeout record is deemed invalid.

5.2.2 Other Procedures: Development, Testing, and Maintenance

We chose a PKI model for key management, so that its established practices and procedures can be used. The use of the root CA's and the VoteCA' authorization keys is only granted to the highest rank staff of the EA (in Brazil, Supreme Court judges preside the Supreme Electoral Court), audited (cryptographically) by political parties, Congress and society representatives.

6. CONCLUSION AND FUTURE WORK

In this paper we propose T-DRE, a trusted computing base for direct recording electronic voting machines, which is mostly independent of the voting application and largely VVSG-compliant. T-DRE's novel combination of technologies enable device verifiability by humans, deep PKI integration and simple auditing. Our architecture was prototyped and then reengineered for large scale manufacturing, with 165,000 devices produced. These DREs will be used in the Brazilian 2010 presidential election.

T-DRE's main component, the Master Security Module (MSM), unifies the TPM and SM modules proposed in the VVSG and adds key new features by: a) enforcing, over the entire software stack, a policy of multi-level, certificate-based access to peripherals and key material; and b) taking control of human interface devices, thus amplifying vote privacy and user DRE tamper detection.

We also indicate how the new audit and control mechanisms present in our architecture can be integrated into the usual electoral cycle, the voting itself, election simulation, device testing and servicing, and software development.

Currently, we are working on the design of a fully-auditable secure processor to be used as a CPU-MSM for DREs.

7. REFERENCES

[1] R. Anderson, M. Bond, J. Clulow, and S. Skorobogatov. Cryptographic processors—a survey. *Proceedings of the IEEE*, 94(2):357–369, 2006.

[2] Brazilian Superior Electoral Court (TSE). Election statistics, April 2010.

[3] D. Chaum. Secret-ballot receipts: True voter-verifiable elections. *IEEE Security & Privacy*, 2(1):38–47, 2004.

[4] B. Chen and R. Morris. Certifying program execution with secure processors. In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 23–23, Berkeley, CA, USA, 2003. USENIX Association.

[5] M. Clarkson, S. Chong, and A. Myers. Civitas: A secure voting system. 2007.

[6] V. Costan, L. F. Sarmenta, M. van Dijk, and S. Devadas. The Trusted Execution Module: Commodity General-Purpose Trusted Computing. In *CARDIS '08: Proceedings of the 8th IFIP WG 8.8/11.2 International Conference on Smart Card Research and Advanced Applications*, pages 133–148, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] C. G. Diego Aranha. Relic is an efficient library for cryptography. http://code.google.com/p/relic-toolkit/, April 2010.

[8] R. Gallo, H. Kawakami, and R. Dahab. On device identity establishment and verification. In *Proc of EuroPKI'09 Sixth European Workshop on Public Key Services, Applications and Infrastructures*, September 2009.

[9] R. Gennaro, A. Lysyanskaya, T. Malkin, S. Micali, and T. Rabin. Algorithmic Tamper-Proof (ATP) Security: Theoretical Foundations for Security against Hardware Tampering, 2004.

[10] A. Huang. Keeping Secrets in Hardware: The Microsoft XBox TM Case Study. *Cryptographic Hardware and Embedded Systems-CHES 2002*, pages 355–430, 2002.

[11] International Organization for Standardization (ISO). *ISO/IEC 11889:2009 Information technology – Trusted Platform Module*. ISO/IEC, 2009.

[12] M. Joye. *Basics of Side-Channel Analysis*, pages 365–380. Cryptographic Engineering. Springer, 1 edition, 2009.

[13] Maxim Integrated Products Inc. Usip-pro component datasheet, April 2010.

[14] D. M'Raihi, M. Bellare, F. Hoornaert, D. Naccache, and O. Ranen. RFC 4226: HOTP: An HMAC-based one-time password algorithm, December 2005.

[15] D. M'Raihi, S. Machani, M. Pei, and J. Rydell. RFC draft: TOTP: Time-based one-time password algorithm, January 2009.

[16] C. Neff. A verifiable secret shuffle and its application to e-voting. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, page 125. ACM, 2001.

[17] C. A. Neff. Practical high certainty intent verification for encrypted votes, October 2004.

[18] NIST. *Security requirements for cryptographic modules, Federal Information Processing Standards Publication (FIPS PUB) 140-2*, 2002.

[19] E. Oksuzoglu and D. Wallach. VoteBox Nano: A Smaller, Stronger FPGA-based Voting Machine (Short Paper). *usenix.org*, 2009.

[20] E. Rescorla. Understanding the security properties of ballot-based verification techniques. In *Electronic Voting Technology Workshop / Workshop on Trustworthy Elections*, August 2009.

[21] R. L. Rivest and J. P. Wack. On the notion of "software independence" in voting systems. *System*, 2006.

[22] P. Rohatgi. *Improved Techiniques for Side-Channel Analysis*, pages 381–406. Cryptographic Engineering. Springer, 1 edition, 2009.

[23] D. R. Sandler. *VoteBox: A tamper-evident, verifiable voting machine*. PhD thesis, Rice University, April 2009.

[24] N. K. Sastry. *Verifying security properties in electronic voting machines*. PhD thesis, University Of California, Berkeley, 2007.

[25] K. Shimizu, H. P. Hofstee, and J. S. Liberty. Cell broadband engine processor vault security architecture. *IBM J. Res. Dev.*, 51(5):521–528, 2007.

[26] G. E. Suh, C. W. O'Donnell, and S. Devadas. Aegis: A single-chip secure processor. *IEEE Design and Test of Computers*, 24(6):570–580, 2007.

[27] The Common Criteria Recognition Agreement. Common criteria for information technology security evaluation v3.1 revision 3, July 2009.

[28] USA Election Assistance Commission. Recommendations to the EAC voluntary voting system, guidelines recommendations, 2007.

Hardware Assistance for Trustworthy Systems through 3-D Integration

Jonathan Valamehr[†], Mohit Tiwari[‡], and Timothy Sherwood[‡]
[†]Department of Electrical and Computer Engineering
[‡]Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
{valamehr@ece,tiwari@cs,sherwood@cs}.ucsb.edu

Ryan Kastner
Dept. of Computer Science and Engineering
Univ. of California, San Diego
La Jolla, CA 92093
kastner@cs.ucsd.edu

Ted Huffmire, Cynthia Irvine, and Timothy Levin
Dept. of Computer Science
Naval Postgraduate School
Monterey, CA 93943
{tdhuffmi,irvine,levin}@nps.edu

Abstract

Hardware resources are abundant; state-of-the-art processors have over one billion transistors. Yet for a variety of reasons, specialized hardware functions for high assurance processing are seldom (i.e., a couple of features per vendor over twenty years) integrated into these commodity processors, despite a small flurry of late (e.g., ARM TrustZone, Intel VT-x/VT-d and AMD-V/AMD-Vi, Intel TXT and AMD SVM, and Intel AES-NI). Furthermore, as chips increase in complexity, trustworthy processing of sensitive information can become increasingly difficult to achieve due to extensive on-chip resource sharing and the lack of corresponding protection mechanisms. In this paper, we introduce a method to enhance the security of commodity integrated circuits, using minor modifications, in conjunction with a separate integrated circuit that can provide monitoring, access control, and other useful security functions. We introduce a new architecture using a separate control plane, stacked using 3-D integration, that allows for the function and economics of specialized security mechanisms, not available from a co-processor alone, to be integrated with the underlying commodity computing hardware. We first describe a general methodology to modify the host computation plane by attaching an optional control plane using 3-D integration. In a developed example we show how this approach can increase system trustworthiness, through mitigating the cache-based side channel problem by routing signals from the computation plane through a cache monitor in the 3-D control plane. We show that the overhead of our example application, in terms of area, delay and performance impact, is negligible.

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

1. INTRODUCTION

The development effort required to build a system is directly proportional to the cost of its failure; hence critical systems used in space shuttles and banks undergo much more rigorous development cycles than systems for home users. *High assurance* systems, which are designed to withstand attacks by professional, well-funded adversaries, require a tremendous investment of time, effort, and money by their small user base. In comparison to commodity systems, these systems generally lag far behind in performance and programmability. Unfortunately, for commodity processors, security threats are often not considered at the rapidly changing ISA [8] or micro-architecture levels. Clearly, a method that allows commodity parts to be retrofitted with protection mechanisms *without increasing the cost* for ordinary users and *without decreasing the performance* of the commodity processor will offer a significant advantage for high assurance system development.

Economics of Hardware Trust: The economics of trustworthy system development puts designers under constraints not faced by low assurance, commodity systems. For example, the expense of special-purpose hardware can make it costlier to provide both high performance and strong security. Even when hardware vendors incorporate security enhancements, integrating these mechanisms into a complex system design may present many practical and theoretical problems, driving up the costs and driving out the release schedule. In addition to the fact that such system devel-

opment costs per unit are very high, users requiring such functionality make up a small portion of the market. Sophisticated security mechanisms at the hardware level are typically targeted at a relatively small market sector and add unacceptable costs to commodity products.

Performance Ramifications: The design cycle of trustworthy systems also places constraints on the performance that can be realized in the final version of these systems. Due to the high non-recurring engineering (NRE) cost of manufacturing custom hardware and the small amortization base of low volume products, manufacturers are often forced to choose less costly alternatives, such as an older, cheaper process (e.g., 0.5um vs. 45nm).

As a result of these economic factors, designers of trustworthy systems requiring high performance need some way to incorporate commercial hardware components without compromising security. To address this challenge, a method of bridging the gap between cutting-edge technology and trustworthy systems is of paramount necessity.

3-D Integration for High Assurance: The primary goal of this paper is to introduce a new method by which security functionality can be added to a processor as a foundry-level configuration option. Specifically, we propose a new and modular way to add security mechanisms to current and next-generation processors through the use of 3-D integration. We advocate consolidating these security mechanisms into a physical overlay, *literally a separate plane* of circuitry stacked on top of a commodity integrated circuit. The security mechanisms that reside in this overlay can then be connected to the underlying chip with a variety of interconnect technologies, yet can be completely omitted without change to the commodity chip's function and without affecting its cost. Using 3-D hardware to alleviate this problem offers many advantages over other hardware solutions as well as software solutions. These advantages are fully explored in Section 2.

Contributions: In this paper, we show that an active layer[1], which we call a *3-D control plane*, specifically dedicated to security, has the potential to implement a variety of security functions in a cost-effective and computationally efficient way. Specifically, this paper makes the following contributions:

- We are the first to develop a method of using 3-D integration for trustworthy system development, and propose to combine an independently fabricated 3-D control plane containing arbitrary security functions (such as micro-architectural protection mechanisms) along with a commodity integrated circuit, which we refer to as the computation plane.
- Security functions can be broadly classified as either active or passive monitors, depending upon whether the 3-D control plane modifies signals on the computation plane. We describe precise circuit-level primitives required to build both active and passive monitors such that signals on the computation plane can be arbitrarily tapped, disabled, re-routed, or even overridden. We also outline how the 3-D control plane can be integrated in a purely optional and minimally intrusive manner with very minor modification to the commodity computation plane.
- We demonstrate our circuit-level primitives using an active monitor that implements a well-known micro-architectural protection mechanism: a cache monitor that can prevent access-driven cache side channel attacks.
- Finally, we validate the functionality of our circuit-level primitives using SPICE simulations, and build a synthesizable prototype of our 3-D cache monitor to evaluate the area-delay cost of its inclusion. We also quantify the impact of our cache protection mechanism on the performance of SPEC benchmark programs, through detailed timing simulations on an out-of-order CPU simulator.

Before describing the circuit-level modifications required of the computation plane, we begin with a discussion of 3-D integration and the opportunities it presents for trustworthy system design.

2. MOTIVATION FOR 3-D SECURITY

In this section, we provide a short background on 3-D integration and present our motivation for using 3-D hardware to address the concerns raised in Section 1. Since 3-D integration is an existing technology already used in industry [24, 26], our work does not discuss the feasibility of 3-D integration but rather focuses on the security ramifications of a 3-D control plane.

2.1 3-D Integration

While the details of how we use this technology are more fully described in Section 3, the main idea is that two pieces of silicon are fused together to form a single chip. The two active layers of the silicon (the commodity computation plane and 3-D control plane) are connected through inter-die vias[2] (micron-width wires that are chemically "drilled-and-filled" between the layers) which run vertically between them. This ability to interconnect multiple active layers enables the addition of an optional die that specifically implements security functions to a commodity processor die. This 3-D control plane would have access to the security-dependent signals of the system. Such a system could be sold to customers requiring application-specific security policy enforcement, information flow control, or other security-specific support. Commodity systems, on the other hand, are unlikely to include this additional, more costly functionality that only benefits a small number of customers.

Attaching multiple layers of silicon together in 3-D stacks is a relatively new, yet already marketed technology [26], which is being explored by most of the major microprocessor manufacturers [6]. As opposed to most current 2-D circuits, which use only one active layer for computation, 3-D circuits contain multiple active layers, or *planes*, which are then connected using techniques such as inter-die vias (or "posts"). Several 3-D interconnect technologies are currently being evaluated in industry as a means of stacking multiple chips together. Some potential applications include the

[1]The active layer is the silicon layer where transistors reside, and metal layers are fabricated above that connect the transistors together. We define a "plane" as the combination of the silicon and metal layers that compose a typical 2-D integrated circuit.

[2]Vias are physical connections between two wires on different metal layers.

Security Architecture	Power	Bandwidth	Delay
Security Functions On-chip	Low power consumption, with the only addition being the power used by security logic and interconnect	Bus width is limited due to contending traffic and component congestion throughout the chip (1-16 bytes) running at core clock speed (>2 GHz)	On-chip delay is dictated by the length of interconnect, which is often very large between components
Security Functions on a Co-Processor	In addition to powering another chip, driving long off-chip bus wires consumes large amounts of power	Low data bus widths due to I/O pin availability (1-8 bytes) running at external clock speed (~ 400 MHz)	Very large delay between off-chip co-processor and CPU (>200 cycles)
Security Functions on a 3-D Control Plane	3-D security only slightly increases power consumption, and can use less power than on-chip due to exploitation of locality of security modules	3-D allows bus widths to be increased significantly (up to 128 bytes) running at core clock speed (>2 GHz)	3-D exhibits low delay due to the short length of inter-die vias, as well as the locality that can be exploited to shorten critical paths

Figure 1: This table compares other hardware options for security against a 3-D control plane and shows the advantages and disadvantages in terms of power, bandwidth, and delay [11, 20].

stacking of DRAM or bigger caches directly onto the processor die to alleviate memory pressure [17] and designing stacked chips of multiple processors [2].

Toshiba has applied 3-D integration to a CMOS image sensor camera module for mobile phones, which they call a Chip Scale Camera Module (CSCM), achieving a significant reduction in size while satisfying high-speed I/O requirements [24]. The Toshiba work demonstrates that cost savings are possible with 3-D integration because passive components, which provide load matching between the chip and the camera, can be integrated into the chip. This makes the passive components cheaper, smaller, and faster than board-level components; therefore, savings can be realized in power, resistance, and capacitance, as driving lines between layers consumes much less power than between chips. Furthermore, multiple layers, each optimized for its particular function, can be combined into a single stack.

Large microprocessor manufacturers are unlikely to integrate support for highly specialized security mechanisms because the market for such features represents such a small portion of their total customer base. This is an example of Gresham's Law: if a manufacturer incurs the cost of security mechanisms deemed unnecessary by the general commodity market, a competing, less costly product without such mechanisms will dominate. By fabricating the optional 3-D control plane with functions that are complementary to (but separate from) those of the main processor, stacked interconnect offers the potential to add security mechanisms to a small subset of devices without impacting the overall cost of the commodity processor.

Just to be clear, we are advocating the development of a processor which is *always fabricated with special connections built in for joining it with a control plane.* The difference between the system sold for the cost-sensitive consumer market and the one that is sold to the security-sensitive customer is only whether a specialized security device is actually stacked on top of the standard integrated circuit, utilizing the special connections. Additional benefits to this approach are that security mechanisms implemented in hardware are faster than software-only approaches, and the security mechanisms can be specialized for particular sets of applications, systems, and customers.

2.2 3-D vs. Other Hardware Solutions

This section discusses the advantages of using 3-D integration over other hardware methods such as on-chip and co-processor implementation of security functions. In general, implementing security functions in software is less costly than in hardware, but software implementations have worse performance and are more susceptible to tampering. Implementing security functions in hardware is more expensive, but the result has better performance and is more resilient to manipulation.

Why not On-Chip?: Implementing security features on-chip creates many issues and discrepancies. It would force all users of the chip uninterested in system trustworthiness to incur the possible negative effects of the added security logic. As discussed previously, an unacceptable consequence of on-chip security is the increase in cost for all consumers. In addition, on-chip security functions have the potential of decreasing the overall performance of the chip, as security modules may need long interconnect wires to data and control lines spanning the whole chip area; this can be mitigated by the exploitation of locality in the 3-D layer as well as short interconnect through inter-die vias as explained in Figure 1. The large majority of microprocessor consumers are chiefly concerned with the performance of the chip, and on-chip security could provide advantages to competing chip manufacturers who do not incorporate these security features. Because of market pressures, chip manufacturers are reluctant to pursue such a course. With 3-D security, the small percentage of consumers who need the added security logic have the option of including it in their systems, while consumers who do not need this extra logic can omit it.

Why not use a Co-processor?: A co-processor solution, much like 3-D security, allows the consumer to have the option of including additional security logic. However, unlike 3-D security, an off-chip co-processor can not safely access internal micro-architectural control signals without possibly making them susceptible to outside tampering. This makes 3-D security much more attractive and feasible, as any resource or control signal can be accessed and modified by the 3-D control plane. Also, co-processor solutions suf-

fer from the utilization of slow, power-hungry off-chip buses. These off-chip buses operate at much slower frequencies than can be realized with a 3-D solution (Figure 1), and they can introduce large delays in processor speed. In addition, off-chip buses have to interface with the main processor through the main processor's I/O pins, and they are limited in size based on available pins. This equates to smaller bus widths (Figure 1), which can further hinder performance. Choosing which pins to interface between the processor and the co-processor also creates inflexible co-processor designs, because we are limited to accessing or modifying those pins, whereas with a 3-D solution we can create any number of different co-processor designs and access any internal signal. Aside from performance, a co-processor solution also entails increased power usage, as driving long off-chip buses requires much more power than driving short inter-die vias to a 3-D control plane. A 3-D security scheme does not fall victim to any of these issues.

Disadvantages of 3-D Security: 3-D security holds much promise as a solution; however, it is not without tradeoffs. Chips fabricated using 3-D integration need greater thermal management, and, without additional cooling, will run at higher temperatures due to the proximity of components [11]. While this is a known issue, it is not insurmountable and can be addressed with more expensive cooling solutions. Another disadvantage of 3-D chips is their expected manufacturing yield, as the functionality of the complete chip is dependent on the individual yield of each of the two dies.This can create lower overall yield than the individual dies. However, the cost of this lower yield will not be incurred by most consumers, as the decrease in yield only applies to the systems that need the 3-D control plane attached.

This section has compared 3-D security with other software and hardware solutions for trustworthy systems. The 3-D control plane can include different types of security monitors. In the next section, we will discuss both of these types of monitors, and follow with our novel circuit architecture to allow the use of an optional 3-D control plane.

3. 3-D SECURITY ARCHITECTURE

The 3-D control plane can include several security functions on one die, implemented as either passive or active monitors. While passive monitoring in 3-D for system profiling has been explored previously [12], a novel contribution of this work is providing active monitoring in a 3-D control plane. In the following section we explain the uses of these two types of monitors, and describe a novel circuit-level architecture that allows us to make the functions of these monitors available as a fabrication option in an overlay.

3.1 Passive and Active Monitors

Passive Monitors: One potential use of the 3-D control plane is to act as a passive monitor, simply accessing and analyzing data from the computation plane. For instance, we may wish to monitor accesses to a particular region of memory or audit the use of a particular set of instructions. To monitor these events, we must understand when such events are occurring, which necessitates *tapping* some of the wires from the processor. This requires posts and vias to the instruction register and memory wires, which gives us direct access to the currently executing instruction.

Passive monitoring is reasonably straightforward to implement in 3-D technology, as it just requires a set of vias to the top of the computation plane, and then a post from there to the 3-D control plane. Figure 2 shows such a post.

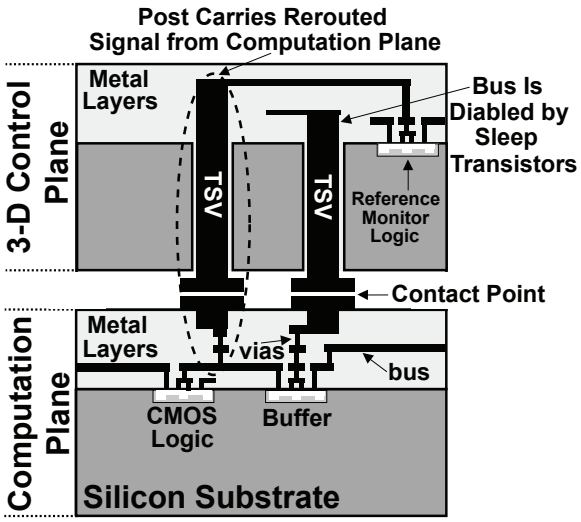


Figure 2: This figure shows the low level architecture for a method to route data/control lines on the computation plane through the 3-D control plane. This can be performed to isolate resources in the computation plane by disabling a bus, for example. The computation plane and the 3-D control plane are connected by inter-die vias or through-silicon vias (TSVs). Posts are required to tap the required signals needed by the security logic, and sleep transistors are used to either reroute, override, or disable lines on the computation plane. Using these primitives, we can build mechanisms to monitor the computation plane.

The area overhead of this passive style monitoring in a 3-D layer was analyzed by Mysore et al. [12] in the context of hardware support for analyzing the processor in real time for debugging and performance profiling, which has high throughput requirements and is very slow to implement in software. Their conclusion was that, even with very pessimistic assumptions about the technology, there would be less than a 2% increase in the total area on the computation plane and that there would be no noticeable delay added. The small amount of area overhead is due to the need to save space for the vias across all of the layers of metal.

Active Monitors: Whereas passive monitoring allows for auditing, anomaly detection, and the identification of suspicious activities, systems enforcing security policies often require strong guarantees about restrictions to overall system behavior. A novel contribution of our work is the employment of active monitors; an active monitor enables control of information flow between cores, the arbitration of communication, and the partitioning of resources.

The key ability needed to support such functionality is to *reroute* signals to the 3-D control plane and then *override* them with potentially modified signals. With this technology and minor modification of the computation plane, we

can force all inter-core communication, memory accesses, and shared signals to travel to the 3-D control plane, where they are subject to both examination and control. For instance, we can ensure that confidential data being sent between two cores (which are traditionally forced to traverse a shared bus) is not leaked to a third party with access to that bus.

We have developed a method to modify signals on the computation plane that is accomplished in two parts. The first part is to ensure that the monitor has unfettered access to all the signals (tapping), which is, in essence, the same as the passive monitoring scenario described above. The second part is to selectively disable those links, essentially turning off portions of the computation plane (e.g., a bus), or overriding them to inject different values. The difficulty is that we must remove a capability (the connection between two components) only by adding a 3-D control plane (which cannot physically cut or impede that wire). The computation plane must be fully functional without an attached 3-D control plane, yet it needs to be constructed so that by adding circuitry, the targeted capability can be completely disabled. To accomplish this, components in the computation plane must be modified to support active monitoring.

3.2 Circuit-level Modifications

This section introduces the circuit level modifications we will make in order for the 3-D control plane to perform its intended function and for the computation plane to be able to execute in its absence. These primitives are illustrated in Figure 4.

Sleep Transistors: A novel and alternative method for disabling links is to physically impede the connection itself. While this sounds intrusive, we are the first to leverage an existing circuit technique called *power gating* [18] for this application. Support for power gating is added through the addition of *sleep transistors* placed between a circuit's logic and its power/ground connections. The sleep transistors act as switches, effectively removing the power supply from the circuit. The circuit is awake when the transistors are activated by a specific signal, which provides power to the circuit, allowing it to function normally. Alternatively, the sleep transistors can be given the opposite input and turned off, thus disconnecting the power to the circuit, temporarily removing all functionality, and effectively putting the circuit to sleep.

Sleep transistors are traditionally used to temporarily disable unused portions of an integrated circuit, saving power by preventing leakage current [19]; however, their use is also beneficial for providing the isolation an active monitor requires. With only a small amount of added hardware (two transistors and two resistors, shown in Figure 3) and posts for connectivity to the 3-D control plane, we can selectively turn off portions of the computation plane to force adherence to any specific security policy enforced in the control layer. Finally, many modern chips already employ power gating. This reduces the amount of additional hardware necessary to apply our security primitives, since only posts to the 3-D control plane to carry the control signal are required.

In addition to selectively removing power from some components on-chip, sleep transistors may be used to perform several key functions on data and control lines required by active monitors. Sleep transistors can be placed on any link that may need to be disabled or controlled. They can be

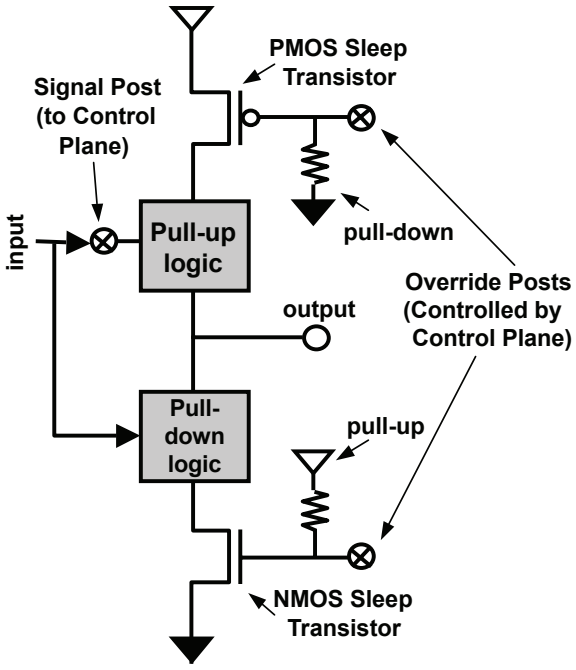


Figure 3: A circuit diagram of sleep transistors in the computation plane being used to remove power from a circuit.

managed by the 3-D control plane by simply providing a post that connects to their gate input. The following functions all use only one or two transistors per line and present a new set of options for trustworthy system development.

Tapping: *Tapping* can be used to send the requested signals to the 3-D control plane without interrupting their original path. As shown in Figure 4a, we use a transistor and apply the correct voltage to the gate of the transistor to create the additional path of the signal to the 3-D control plane. This is particularly useful when we are performing analysis (e.g., dynamic information flow tracking) on the flow of information on the computation plane without affecting its original functionality (Figure 6). *Tapping* can also be used when security logic on the 3-D control plane is dependent on some data in the computation plane, without the need to change their values in the system. In our 3-D cache eviction monitor (Section 3.3) we use *tapping* to access the address of a load or a store instruction to determine whether a cache eviction is allowed without interfering with the normal flow of the address through the bus.

Re-routing: *Re-routing* (Figure 4b) uses two transistors per line to send the requested signals to the 3-D control plane and block their transmission to the originally intended path. A pull-up resistor is attached to the gate of the transistor that is *disabling* the line, to force a connection when the 3-D control plane is not attached. *Re-routing* can be used in situations where we want to create new buses between resources on-chip.

Another use of *re-routing* is using a signal for a different purpose than was originally intended. Once on the 3-D control plane, the signal can be analyzed and combined with other data from the 3-D control or computation planes, or simply stored for later use. This can then be coupled with *overriding* (Figure 4c) to change control or data outputs on

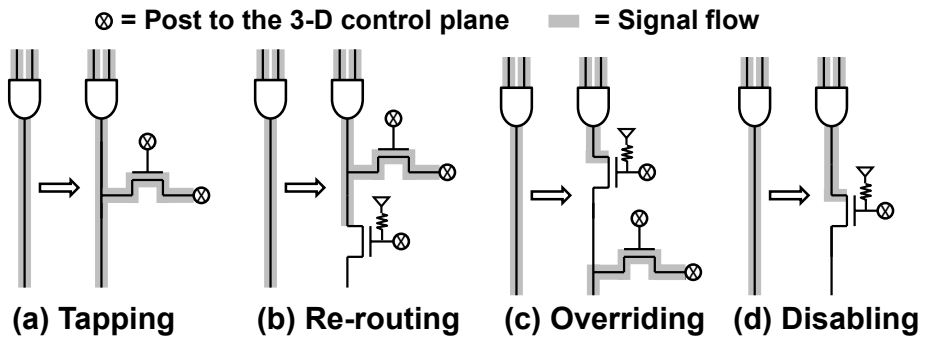


Figure 4: This figure shows the four different kinds of circuit level modifications that can be made and their respective diagrams. The sample base circuit is an AND gate and is found to the left of each circuit modification. *Tapping* requires only one transistor to optionally propagate the signal to the 3-D control plane, while *re-routing* and *overriding* need transistors with pull-up resistors to ensure their continued function for systems omitting the 3-D control plane. *Disabling* uses a transistor and a pull-up resistor to uphold the connection in the absence of the 3-D control plane, while giving the 3-D control plane the option of disconnecting the line for systems utilizing it.

the computation plane based on new logic in the 3-D control plane (Figure 7).

Overriding: *Overriding* (Figure 4c) allows us to block the intended value of a signal and modify it to a desired value for the security layer's function (Figure 6 and Figure 7). *Overriding* uses two transistors and a pull-up resistor much like *re-routing*. For some security applications, critical control signals need to be changed in order to adhere to a security policy that is being enforced by the 3-D control plane. In our 3-D cache eviction monitor (Section 3.3), we use *overriding* to change the value of a cache's write-enable signal (see Figure 8), allowing us to inject a value to allow or deny the eviction of a specific cache line.

Disabling: *Disabling* (Figure 4d) allows us to completely stop the flow of data on a common bus or a specific signal line. Uses of disabling include the ability to isolate a specific resource from unintended accesses, or enforcement of policies that require tight guarantees on the integrity of data on a shared bus. Many bus protocols work on a *mutual trust* system, where access to the bus is controlled by the devices that are connected, not by a trusted arbiter. In situations such as this, it is important to preserve trustworthy execution and the confidentiality of data during a sensitive computation. *Disabling* can be used to forcibly block access to a bus to ensure secure transactions without the possibility of unintended access (Figure 5).

3.2.1 Spice Simulation Results:

To verify the correctness of our circuit-level modifications, we developed Spice circuit models for each of the circuits in Figure 4 and used Spice simulations to read the voltage values at certain nodes for each circuit. Two experiments were performed, with input voltages at the transistor terminals corresponding to the 1) absence of the 3-D control plane and corresponding to the 2) presence of the 3-D control plane. NMOS transistors from 45nm predictive technology models [1] were used to characterize the sleep transistors, but PMOS transistors can also be used. Regardless of which transistor type we use, we need to buffer the signal after it has traveled through the transistor to ensure a strong signal propagation. During the experiment where the 3-D control plane is omitted, the transistor gates are not powered, and the pull-up resistors successfully power the transistor's gate and create a short, allowing the signal to pass normally. When the transistor gates are powered, we can successfully control the circuit and perform the function for each respective circuit. These experiments verify our ability to create functional systems with the option to add a modular 3-D control plane.

3.3 Theoretical 3-D Applications

Isolation: One potential application of our circuit-level primitives is the active isolation of resources in a system. For example, in multi-core processors there are shared data and address buses that rely on a mutually trusting shared bus protocol, where each core is responsible for its own arbitration. This is problematical for the security of bus traffic on a system running code of varying trust levels on each core. Figure 5 outlines this situation and how we can use *Disabling* to disconnect a core from the bus for any given amount of time, creating a Time Division Multiple Access (TDMA) protocol between the cores and the shared resources of interest.

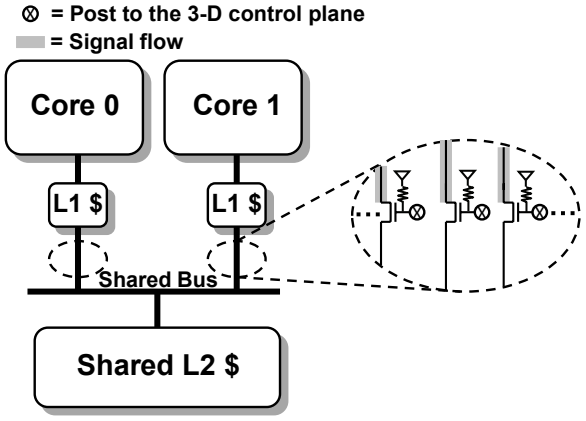


Figure 5: A multi-core processor with two cores that we wish to isolate. This is achieved using *Disabling* to block the connections to the bus for the core that is not currently allowed to use the bus.

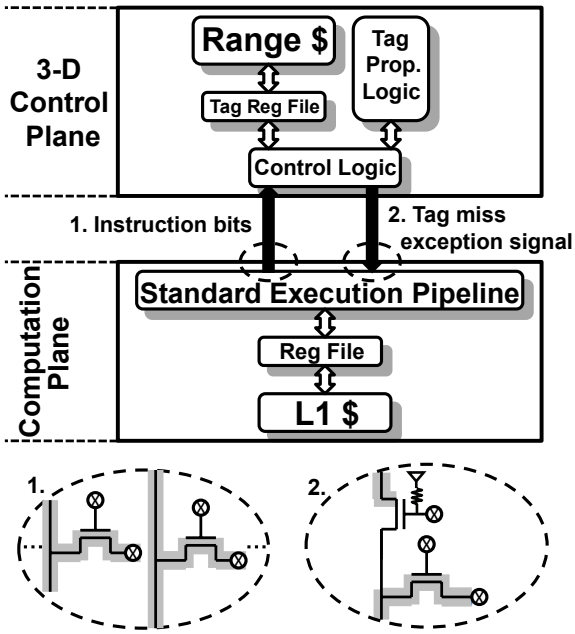


Figure 6: A 3-D system monitor tracking data flow on the computation plane. This is achieved using *Tapping*(1) to read signals that we want to track and *Overriding*(2) to raise an exception signal.

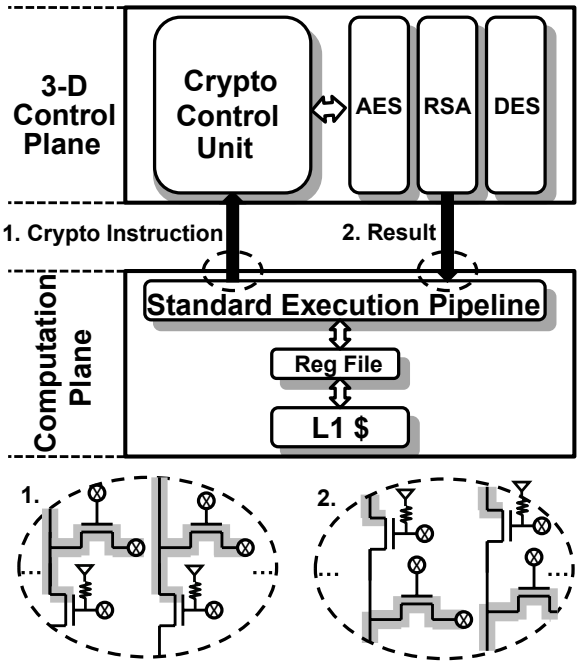


Figure 7: A 3-D cryptographic engine used to perform secure cryptography in the 3-D control plane, using *Re-routing*(1) to block the instruction execution on the computation plane and to send the instruction to the 3-D control plane to be executed. The result can be placed back in the execution pipeline using *Overriding*(2).

System Analysis and Monitoring: It is often useful to monitor the activity of the computation plane for auditing, intrusion detection, or post-mortem analysis. Information flow tracking in the 3-D control plane, for example, attempts to identify, track, mitigate, and deter the execution of malicious code. The basic premise of dataflow tracking is the storage of *metadata* in the form of tags associated with each individual address in memory. A dataflow tracking architecture with a small cache [21] that compresses memory addresses with matching metadata tags can be utilized in the 3-D control plane (Figure 6), to raise an exception in the event that malicious execution on the computation plane is detected. For such a monitor, we can use *Tapping* to read signals of interest on the computation plane and use *Overriding* to optionally modify an exception signal without tampering with normal use.

Secure Alternate Service: Another potential application is augmenting the functionality of the computation plane with additional hardware for security computations. For systems requiring high-bandwidth cryptographic functionality, we can implement a cryptographic engine on the 3-D control plane that can accept cryptographic instructions being executed on the computation plane, performing the operation immediately before sending the result back to the execution pipeline. This is achieved by using *Re-routing* to extract the cryptographic instructions from the standard execution pipeline, execute the instruction, and use *Overriding* to inject the result into the pipeline as if it were part of the normal instruction execution flow. While cryptographic hardware has been included in microprocessors [8], 3-D security allows the addition of any cryptographic algorithm or implementation to be included in the system as a foundry-level option. Essentially, 3-D security introduces flexibility in the system hardware, allowing any number of cryptographic cores to be optionally added to the processor.

The techniques described in this section provide powerful tools for implementing active monitors in the 3-D control plane, thereby allowing the addition of security-critical functionality. If used appropriately, this can eliminate certain types of side channels by mediating the use of a shared resource. In the following section we present the architecture of an active cache eviction monitor that we have implemented for the 3-D control plane using the previously discussed circuitry.

3.4 Architecture of a 3-D Cache Monitor

This section presents the custom architecture shown in Figure 8, implemented in the 3-D control plane, for eliminating access-driven cache side channel attacks. Concurrent processing platforms present several security issues; although these architectures provide increased performance through instruction-level parallelism, their methods of resource sharing leave them vulnerable to side channel attacks. One side channel attack [16] uses a simultaneous multithreading processor's shared memory hierarchy, exploiting the process-to-process interference arising from the cache eviction policy to covertly transfer information. As a result, an attacker thread may be able to extract information from a victim thread, such as a cryptographic key. This threat was demonstrated by Percival [16], where an implementation of the RSA encryption standard was attacked using the cache eviction protocol and used to observe, in small chunks, the total cryptographic key. This was achieved by having a ma-

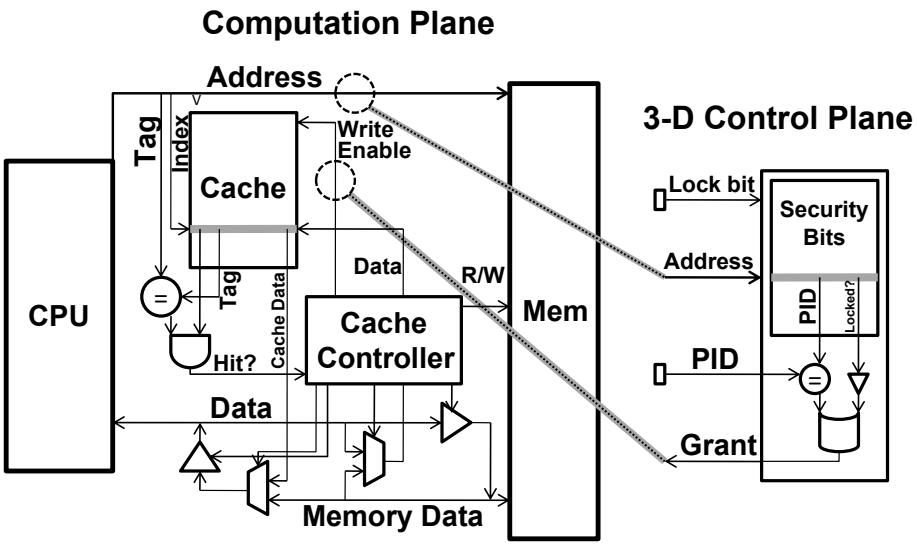


Figure 8: The architecture of a CPU/cache memory hierarchy and our 3-D cache eviction monitor working in concert. The address of the corresponding load/store is tapped to be sent to the 3-D control plane, and the cache write-enable signal is overridden in the case of a locked cache line eviction. The Lock bit as well as the Process ID (PID) are also provided to the 3-D control plane. We discuss options on how to access this information in Section 4. Once the cache monitor receives the load/store address, the Lock bit, and the PID, it can determine whether a cache eviction can be granted based on whether the cache line is locked or whether the PID matches, and issue the appropriate *override* signal on the cache write-enable signal.

licious thread consume sufficient memory so that when the victim thread executed, the spy thread's cache lines would be evicted. Thus by measuring subsequent access times for its cached items, the spy thread can observe which of its cache lines had been evicted by the victim. Once the spy thread knows these cache lines, it can infer parts of the cryptographic key due to the nature of the table look-ups performed during the encryption. Slowly but surely, the whole key can be compromised with a relatively low margin of error.

Our method to prevent these attacks is based on a previously proposed hardware solution [23]. In our application of this scheme, the 3-D control plane maintains a cache protection structure that indicates, for each cache line, whether it is protected, and if so, for which process. When a different process loads or stores data related to a protected cache line, no eviction will occur, and the data is not cached unless an alternate line is available in the cache protocol being used. Figure 9 shows a flowchart describing this new protocol, while Figure 10 provides a high-level overview of how the cache and the 3-D control plane will interact. Specifically, the cache protection structure contains memory elements on the 3-D control plane to store *security bits*, which hold the permissions of a process to evict shared cache entries of other processes. With this in place, when instructions proceed to load or store data, these security bits are first checked to determine whether to grant a cache eviction that might otherwise have occurred without policy oversight. As mentioned previously, when the 3-D control plane is not attached to the processor, the cache functions as normal. However, when the 3-D control plane is added, we can utilize the above strategy to avoid undesirable cache evictions. This is performed with an updated version of the *load* and *store* instructions. These instructions, named *secure_load* and *secure_store*, change the security bits in the 3-D control plane to reflect the process that currently occupies the line. Effectively, *secure_load* and *secure_store* modify the necessary bits to ensure that once a cache line is occupied by a process that needs cache eviction control, it cannot be evicted by any other process. This will control a simultaneous multithreading processor's shared memory and eliminate any threat of an access-driven side channel attack.

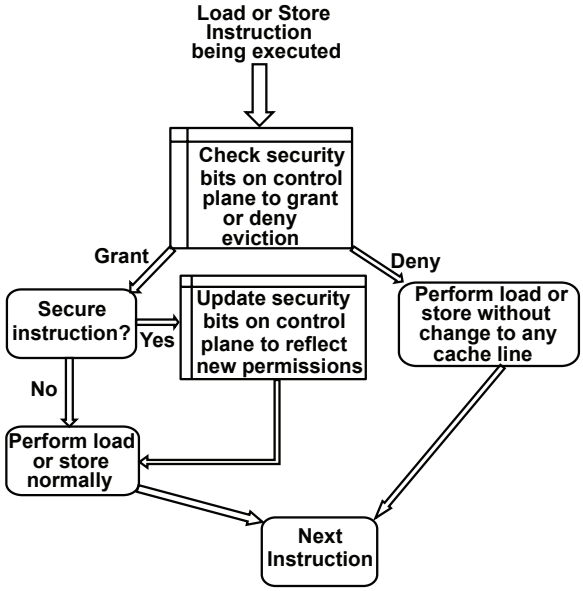


Figure 9: This flow chart describes how loads and stores are executed when the 3-D control plane is in place.

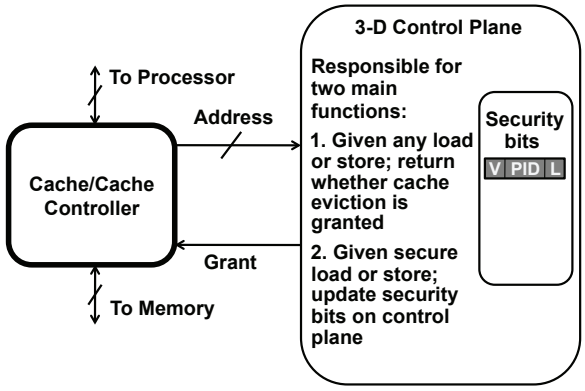


Figure 10: A high-level logical overview of how the cache and the 3-D control plane interact in our cache monitor, as well as the 3-D control plane's responsibilities when active.

As a proof of concept, we have developed a synthesizeable version of our security mechanism in Verilog. We designed our security mechanism as a separate module that is interfaced with a simple cache that we also implemented as a hardware design. Our design uses a straightforward 4-way set associative cache. For every load or store instruction, the cache controller first checks the 3-D control plane module to determine whether the related cache line is protected from evictions. The security bits on the 3-D control plane hold a valid bit, a process ID, and a lock bit for each cache line. During the loads and stores, these security bits are checked in the 3-D control plane, and a *grant* signal is generated if the cache line is open to eviction. While every load and store will be forced to check the security bits before proceeding, these security bits can only be manipulated by using *secure_load* and *secure_store*.

We synthesized both modules and have verified that the design is functional, easily scaled, and can be implemented with low overhead. This will be discussed in further detail in the following sections where we analyze performance metrics, overhead for a modern processor, and feasibility.

4. EXPERIMENTAL RESULTS

This section outlines our synthesis results, and discusses the effect of including the 3-D cache eviction monitor, both in terms of critical path and cache performance. We find that the 3-D cache eviction monitor does not increase the critical path of the circuit, and we observe that this type of cache-line locking produces very little performance degradation for many programs. We also discuss integration options and feasibility for the 3-D control plane on a sample commodity processor.

4.1 Performance and Analysis

Synthesis Results: In this section, we analyze the performance and area overhead of the 3-D cache eviction monitor. We use Altera Quartus to synthesize our design and extract specific timing and area information (Figure 11). To provide a clear picture of the overhead and performance effects of our design, we gathered timing and area information for both the cache/cache controller alone, as well as the cache/cache controller being interfaced with the 3-D cache eviction monitor module. The synthesis was performed for a Stratix II device, with the compiler set to optimize for performance. The standalone cache was able to run at approximately 151MHz; when we include our 3-D cache eviction monitor, the maximum frequency remains at 151MHz. The 3-D cache eviction monitor synthesized by itself has a maximum frequency of 217MHz. These maximum frequencies indicate that the critical path in the circuit including the 3-D cache eviction monitor resides in the underlying cache/cache controller, resulting in no change in cycle time for the circuit with the addition of the 3-D cache eviction monitor.

Design	Max Frequency	Area (LUTs)
Cache/Cache controller	~151MHz	468
3-D cache eviction monitor	~217MHz	291
Cache/Cache controller with 3-D monitor attached	~151MHz	749

Figure 11: The synthesis results produced by Quartus for the cache and cache controller, as well as the 3-D cache eviction monitor.

The above performance metrics do not take into account the delay of the vertical posts between the computation plane and the 3-D control plane. Loi et al. [11] characterized the worst-case delay of a 3-D bus that travels from one corner of a chip to the opposite corner on a 3-D layer above, and they found this delay to be about .29ns. Even with the addition of this bus delay to the 3-D cache eviction monitor's critical path, the new critical path is still less than that of the cache/cache controller, further confirming that the addition of the 3-D cache eviction monitor will have no effect on the performance of the cache subsystem.

Performance Evaluation: We evaluated the performance impact of locking specific cache lines with our 3-D cache eviction monitor. We used PTLsim [25], a cycle-accurate x86 simulator, to execute the SPEC2000 benchmark suite. The experiments we developed outline two scenarios:

- 1) Running each benchmark with a 32KB 4-way set associative cache, representing a 32KB L1 cache with no cache line locking. This is a best-case performance bound because running the benchmark and the AES program together will be slower than running the AES program by itself.
- 2) Running each benchmark on a 32KB 4-way set associative cache, with one of the ways locked, effectively resulting in a 24KB 3-way set associative cache. This is a worst-case performance bound because the AES program is smaller than an entire way of the cache (8192 bytes).

We modeled our cryptographic process after the AES algorithm, which can occupy up to 4640 bytes with an enlarged T-Box implementation [5]. With this in mind, 8192 bytes is

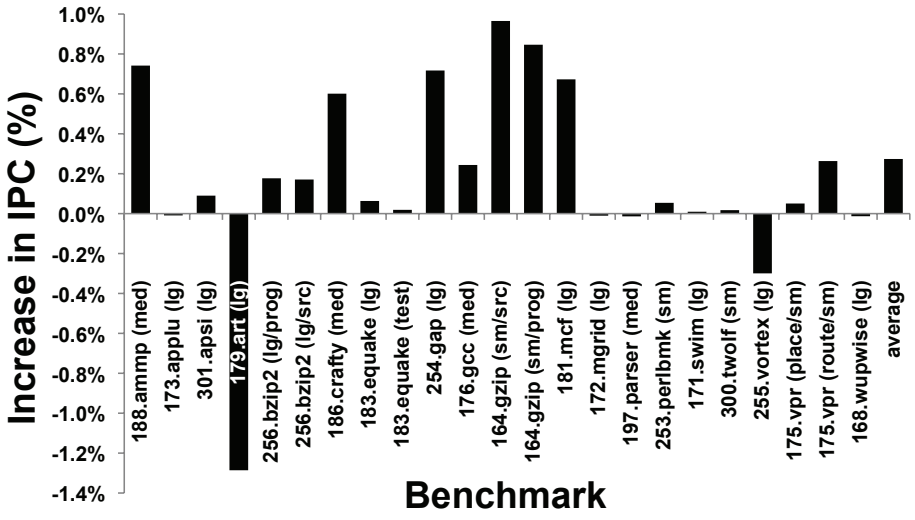


Figure 12: The results of our cache experiment using SPEC2000 benchmarks that were executed in PTLsim. We use two different sizes of cache to calculate a bound on the performance impact of locking a cryptography program like AES to one way of the cache. The average degradation in IPC between the two different cache sizes is 0.2%, indicating that this form of cache line locking has a small impact on performance for many different types of programs.

more than enough to store all of the necessary information (state vectors, round keys, and look-up tables) for AES.

Results for this experiment can be found in Figure 12. The average degradation in IPC is 0.2%, indicating that this form of cache line locking has a small impact on performance for many different types of programs. We were not able to build the binaries for mesa, galgel, facerec, or fma3d. In addition, we encountered technical difficulties with lucas, eon, and sixtrack.

4.2 Discussion and Integration Options

When integrating our security scheme for cache management with a processor, several factors must be considered. Implementing security functionality requires the following capabilities: access to the process ID of a thread during its execution, access to the address bus, and a method of discerning between normal and secure loads and stores. These are the high-level requirements of the 3-D control plane; some vertical posts are also needed to propagate this information to the 3-D control plane.

For our 3-D cache eviction monitor to function, we need to know the process ID of the thread performing the current load or store function. One option we have explored is accessing the process ID register that some architectures have, such as the ARM926EJ-S [10]. Accessing this register through the vertical posts will give the 3-D control plane direct access to the current process ID, allowing the control plane to compare it to the security bits.

We also need to know when loads and stores are being executed. One option is *tapping* the instruction bus, allowing us to monitor the execution of loads and stores and subsequently apply our security functions to those instructions. During the execution of loads and stores, the control plane will follow the protocol outlined in Figure 9.

Finally, the 3-D control plane must know whether each load and store operation is secure or not, so that the system can determine whether the security bits in the 3-D control plane need to be updated. One way to supply this information is to modify the instruction set to include two special instructions, *secure_load* and *secure_store*. This would create separate instructions of which the 3-D control plane is aware in order to distinguish between normal load/store and secure load/store operations. Another option is to add a register to the computation plane that reflects whether the current instruction is secure or not. The operating system can control this bit based on whether the instruction is secure or not, and the control plane could read this register. Both options are feasible and have no negative implications on the rest of the system.

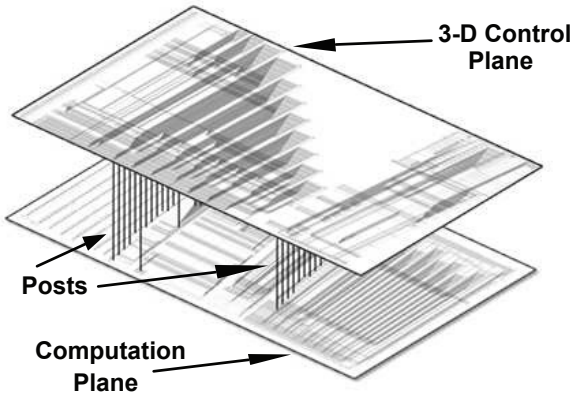


Figure 13: This figure is a visualization of the physical circuit-level diagram of our 3-D cache monitor. Gate-level diagrams were compiled in Quartus after synthesis of the modules.

Delivery of the previously mentioned required information to the 3-D control plane will be through the vertical posts. A general idea of the number of posts the 3-D control plane needs on a given system is the sum of the number of bits of: the *address size*, the *process ID size*, possibly one post for the secure register, and a *grant* bit post. For the ARM926EJ-S, this results in under 100 vias, which equates to about the silicon space for 50 bits of memory; this is a small and reasonable number of vertical posts to implement a strong security measure.

5. RELATED WORK

In this section, we discuss other work associated with cache side channel problems. We also discuss related work on the use of 3-D technology for security and communication as applied to CMP architectures.

On-chip and board-level resource sharing between cores is often used to enhance CMP performance. However, contention for those resources at the microarchitectural level can provide the basis for *side-channel cryptanalysis* attacks and other covert timing channels. Code and data caches, as well as the branch prediction unit, are some of the shared resources that can be exploited in these attacks [9, 4, 3]. In these cases, one process's use of the resource perturbs the response time of the next process that accesses it, in a predictable manner. Single-core computers with simultaneous multithreading, and SMP systems with cache coherency mechanisms, can have similar problems.

One approach to prevent resource contention in a concurrent execution model is to utilize separate physical caches for each core, or provide separate virtual caches within the physical cache (if virtual cache support is available in hardware) [15, 23]. Various forms of cache disablement are possible, including turning it off, turning it off for certain cores or processes, or turning off the eviction and filling of the cache through use of the processor *no-fill* mode. The latter can be used to create *sensitive sections* [13] of code that could not interfere with the cache behavior observable by other cores or processors – assuming that the code is not interruptible or that the previous processor mode is restored on interrupt, as otherwise, other processes might sense the change to the state of the processor (i.e., to *no-fill*), creating another covert channel [5].

Specific cryptographic attacks can be defeated or minimized by lowering the bandwidth of the cache channel, such as through nondeterministic ordering of access to cache [14] which makes detailed cache-use profiling difficult; and nondeterministic cache placement [22, 15] or nondeterministic polyinstantiation [7] of cache entries, [23] which, while the specific cause of the interference may be masked, still allows detection of cache misses caused by another process. The 3-D approach has the advantage of being able to implement many of these schemes for resolving cache contention, while doing it in an isolated environment, without modification to the processor ISA.

6. CONCLUSIONS

3-D integration offers the ability to decouple the development of security mechanisms from the economics of commodity computing hardware. We described the technology to enable passive and active monitoring of the computation plane by adding a minimal amount of hardware. Passive monitoring requires vias and posts, while active monitoring uses sleep transistors to perform several novel functions on the computation plane. Using these techniques, we described a number of broad strategies to enhance the security of the computation plane with a control plane. To provide quantitative measurements of the impacts of the control plane, we considered cache side channels, developing a complete hardware description for a cache with a control plane that eliminates eviction-based side channels. This work provides a pathway for the high-assurance community to utilize high-performance hardware while shortening development cycles for trustworthy systems.

Acknowledgements

The authors would like to thank the anonymous reviewers for their insightful comments. This research was funded in part by National Science Foundation Grant CNS-0910734.

7. REFERENCES

[1] Arizona State University Predictive Technology Models, Predictive Technology Models for 45nm Processes, Available at. `http://www.eas.asu.edu/~ptm/`.

[2] N. G. A. Akturk and G. Metze. Self-Consistent Modeling of Heating and MOSFET Performance in 3-D Integrated Circuits. *IEEE Transactions on Electron Devices*, 52(11):2395–2403, 2005.

[3] O. Acıiçmez. Yet another microarchitectural attack: Exploiting I-cache. In *Proceedings of the First Computer Security Architecture Workshop (CSAW)*, Fairfax, VA, November 2007.

[4] O. Acıiçmez, J. Seifert, and C. Koc. Micro-architectural cryptanalysis. *IEEE Security and Privacy Magazine*, 5(4), July-August 2007.

[5] D. J. Bernstein. Cache-timing attacks on AES. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`, Apr. 2005. Revised version of earlier 2004-11 version.

[6] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCauley, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb. Die Stacking (3D) Microarchitecture. *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–479, December 2006.

[7] D. E. Denning and T. F. Lunt. A multilevel relational data model. In *Proc. IEEE Symposium on Security and Privacy*, pages 220–234, 1987.

[8] S. Gueron. White paper: Advanced encryption standard (AES) instructions set, Intel corporation, July 2008.

[9] J. Kelsey, B. Schneier, C. Hall, and D. Wagner. Side channel cryptanalysis of product ciphers. *Journal of Computer Security*, 8(2–3):141–158, 2000.

[10] A. Limited. ARM926EJ-S technical reference manual, 2001-2008.

[11] G. L. Loi, B. Agrawal, N. Srivastava, S.-C. Lin, T. Sherwood, and K. Banerjee. A Thermally-Aware Performance Analysis of Vertically Integrated (3-D) Processor-Memory Hierarchy. In *Proceedings of the 43nd Design Automation Conference (DAC)*, June 2006.

[12] S. Mysore, B. Agrawal, S. Lin, N. Srivastava, K. Banerjee, and T. Sherwood. Introspective 3-D chips. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, October 2006.

[13] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES: (extended version). Technical report, Department of Computer Science and Applied Mathematics, Weizmann Institute of Science,, Rehovot 76100, Israel, Oct. 2005.

[14] D. Page. Theoretical use of cache memory as a cryptanalytic side-channel. Technical Report CSTR-02-003, Department of Computer Science, University of Bristol, June 2002.

[15] D. Page. Partitioned cache architecture as a side channel defence mechanism. In *Cryptography ePrint Archive, Report 2005/280*, August 2005.

[16] C. Percival. Cache missing for fun and profit. In *Proceedings of BSDCan 2005*, Ottowa, Canada, May 2005.

[17] K. Puttaswamy and G. H. Loh. Implementing Caches in a 3D Technology for High Performance Processors. In *IEEE International Conference on Computer Design (ICCD) 2006*, pages 525–532, October 2005.

[18] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand. Leakage current mechanisms and leakage reduction techniques in deep-submicrometer CMOS circuits. *Proceedings of the IEEE*, 91(2), February 2003.

[19] K. Shi and D. Howard. Sleep transistor design and implementation simple concepts yet challenges to be optimum. *IEEE VLSI-DAT Taiwan*, 2006.

[20] H. Sun, J. Liu, R. S. Anigundi, N. Zheng, J.-Q. Lu, K. Rose, and T. Zhang. 3D DRAM design and application to 3D multicore systems. *Design and Test of Computers, IEEE*, 26(5), September 2009.

[21] M. Tiwari, B. Agrawal, S. Mysore, J. K. Valamehr, and T. Sherwood. A small cache of large ranges: Hardware methods for efficiently searching, storing, and updating big dataflow tags. In *Proceedings of the International Symposium on Microarchitecture (Micro)*, Lake Como, Italy, November 2008.

[22] Topham and Gonzalez. Randomized cache placement for eliminating conflicts. *IEEETC: IEEE Transactions on Computers*, 48, 1999.

[23] Z. Wang and R. Lee. New cache designs for thwarting cache-based side channel attacks. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, June 2007.

[24] H. Yoshikawa, A. Kawasaki, T. Iizuka, Y. Nishimura, K. Tanida, K. Akiyama, M. Sekiguchi, M. Matsuo, S. Fukuchi, and K. Takahashi. Chip scale camera module (CSCM) using through-silicon-via (TSV). In *Proceedings of the International Solid-State Circuits Conference (ISSCC)*, San Francisco, CA, February 2009.

[25] M. T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 23–34, 2007.

[26] I. Ziptronix. 3D integration for mixed signal applications, 2002.

SCA-Resistant Embedded Processors—The Next Generation

Stefan Tillich
University of Bristol
Computer Science Department
Merchant Venturers Building
Woodland Road, BS8 1UB,
Bristol, UK
tillich@cs.bris.ac.uk

Mario Kirschbaum
Graz University of Technology
Institute for Applied Information Processing and Communications
Inffeldgasse 16a, A–8010
Graz, Austria
mkirschbaum@iaik.at

Alexander Szekely
Graz University of Technology
Institute for Applied Information Processing and Communications
Inffeldgasse 16a, A–8010
Graz, Austria
aszekely@iaik.tugraz.at

ABSTRACT

Resistance against side-channel analysis (SCA) attacks is an important requirement for many secure embedded systems. Microprocessors and microcontrollers which include suitable countermeasures can be a vital building block for such systems. In this paper, we present a detailed concept for building embedded processors with SCA countermeasures. Our concept is based on ideas for the secure implementation of cryptographic instruction set extensions. On the one hand, it draws from known SCA countermeasures like DPA-resistant logic styles. On the other hand, our protection scheme is geared towards use in modern embedded applications like PDAs and smart phones. It supports multitasking and a separation of secure system software and (potentially insecure) user applications. Furthermore, our concept affords support for a wide range of cryptographic algorithms. Based on this concept, embedded processor cores with support for a selected set of cryptographic algorithms can be built using a fully automated design flow.

Categories and Subject Descriptors

B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Microprocessors and microcomputers*; K.6.5 [**Computing Milieux**]: Management of Computing and Information Systems—*Security and Protection*; C.3 [**Special-Purpose and Application-Based Systems**]: Smartcards

General Terms

Security

Keywords

Side-channel analysis, SCA countermeasures, embedded processors

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

1. INTRODUCTION

So far, most SCA countermeasures proposed in literature deal either with the protection of dedicated hardware (e.g. cryptographic coprocessors) or of software implementations. However, some works have addressed the issue of integrating protection mechanisms directly into the processor. May *et al.* proposed the concept of non-deterministic processors [9], where the instructions are executed in a more or less random fashion. A potential issue for such non-deterministic processors is the dependency of security and efficiency on the parallelism of the executed code. Regazzoni *et al.* [12] have developed an automated design flow which can integrate custom functional units protected by a secure logic style into a basic processor architecture. While automatization is very desirable, their work so far only addresses a fraction of the problem, as side-channel leakage not only emanates from the processor's functional units but also from several other parts which hold critical data (e.g. pipeline registers and memory). It is currently unclear how their solution can be extended to protect the whole processor.

We have drawn the basic idea for the protection mechanism described in this paper from the paper of Tillich *et al.* which describes SCA countermeasures in the context of cryptographic instruction set extensions [15]. In this concept, a part of the processor is protected by a secure logic style, while the rest of the hardware is left unchanged. One of the key advantages of our solution is that only a fraction of the processor needs to be implemented in the costly secure logic style, whereas a naive approach would incur this implementation overhead for the complete processor system. Note that the naive approach would require also the complete external memory to be implemented with a side-channel resistant memory technology in order to achieve protection equivalent to our solution. The authors are currently not aware of a satisfying solution for implementing large memories with resistance to power analysis which could be used for protecting the external memory. Furthermore, any such solutions are likely to incur a substantial overhead in terms of area and power. Finally, the simple use of standard memory technology for external memory would be prevented. In contrast, our solution protects all values occurring outside of the secure part of the processor (including external memory) with a mask. The masks themselves reside only in the secure part.

This paper consists of three main contributions. First, we

significantly extend the concept from [15] with functionality which allows flexible application in secure multi-tasking operating systems with full process isolation. Second, we present concrete implementations of the proposed protection mechanism with different tradeoffs in a typical embedded processor and provide concrete cost estimations. Third, this paper presents the first practical side-channel evaluation of the secure zone concepts (without secure logic), which demonstrates the soundness of the approach.

With the exception of the implementation of the NONDET processor architecture by Grabher *et al.* [6], this paper is the first to present a concrete implementation of an embedded processor with a generic protection mechanism against side-channel attacks. Our solution is not quite as generic as the NONDET processor, but on the other hand its security is inherently independent from the parallelism of the underlying workload.

Our work differs from other secure processors like the Aegis processor [13] in the model of the attacker. We assume that an attacker has only access to the input and output of the device and can measure its power consumption, but that the contents of the memory is not directly accessible. On the other hand, the Aegis processor is designed to withstand probing and manipulation of the external memory, but not to withstand power analysis attacks. Thus, our proposal is suited for applications where the attacker does not have direct memory access, e.g. devices with secure on-chip storage or tamper-proof casings, whereas the Aegis processor is more suited for applications where power analysis is not considered a threat.

Our security concept is conceived as a building block to enable SCA-resistant implementations if used in conjunction with suitable components like development tools, secure operating systems and/or applications. Furthermore, our concept is compatible with other often-required security features like process isolation. However, we would like to point out that our concept can not safeguard against all possible problems which may arise from an incorrect implementation of other security-related components (e.g. software writing keys to user-accessible interfaces).

The rest of the paper is structured as follows. The basic principles of our proposed countermeasures are described in Section 2, while extended functionality is presented in Section 3. Details and design choices for our prototype implementation are given in Section 4. Section 5 shows how protected processors can be built using an automated design flow and that most of the tasks for administering the secure zone can be offloaded to the compiler. Practical results regarding the hardware size of different implementation options of the countermeasures and a preliminary SCA evaluation are given in Section 6. Conclusions are drawn in Section 7.

2. BASIC CONCEPT

In the context of SCA resistance, critical operations are those which involve data that can be exploited in an SCA attack. For DPA attacks, this are typically intermediate values of cryptographic algorithms which depend on a small portion of the key and a part of the input or output. For example, some of the round transformations of an AES encryption would be critical operations. In processors equipped with our protection mechanism, all critical operations are exclusively executed within the boundaries of a single hardware

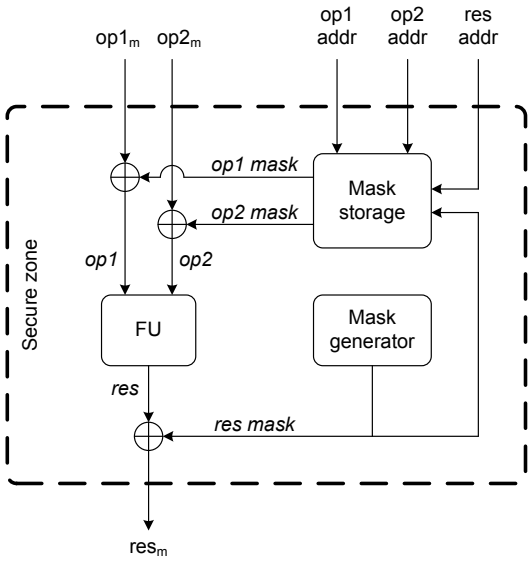


Figure 1: Overview of the basic components of the secure zone.

sub-module (the so-called *secure zone*). The secure zone is implemented in a secure (DPA-resistant) logic style, while the rest of the processor is implemented in standard CMOS. For the processor, the secure zone behaves like any other ordinary functional unit[1]. It offers a range of instructions which are useful for implementing cryptographic algorithms. An overview of the components of the secure zone is depicted in Figure 1.

All instruction operands and results outside of the secure zone are masked. The corresponding masks are held exclusively in the *mask storage* component within the secure zone. The masks can be retrieved from the mask storage and removed from the operands yielding the unmasked operands $op1$ and $op2$. The *functional unit* (FU) produces the result res of the current processor instruction. The *mask generator* outputs a fresh mask which is applied to the result (yielding res_m) and written to the mask storage. The masked result then leaves the secure zone and is handled by the processor like any other "normal" register value, *i.e.* it can be written to memory or can be used as operand in subsequent instructions.

Many previous masking solutions manipulate masked values and compensate for the change of the mask afterwards. Note that this is not the case for our countermeasure, where we always use a fresh mask whenever a masked value is updated by the cryptographic algorithm. Thus there is a total independence of the performance of cryptographic operations and the applied masks. Hence, no special effort is necessary to cater for the protection of non-linear cryptographic operations, which is usually an issue in traditional masking countermeasures.

Some form of addressing mechanism is required to asso-

[1]A functional unit of a processor is a hardware module within the execute stage which takes a number of operands and produces the according result for a single or a range of instructions. Typical functional units include adders, multipliers, barrel shifters, and modules implementing bitwise logical functions.

ciate masked operands and masks during operation of the processor. A simple example for such an addressing mechanism is the employment of the operands' register addresses. The addresses for the masked input operands $op1_m$ and $op2_m$ are used to retrieve the corresponding masks from the mask storage. Similarly, the address of the instruction's masked result needs to be associated with the corresponding mask, so that the masked result can be used as an instruction operand in subsequent instructions.

As the masks never leave the secure zone, the power consumption of the CMOS part should only contain leakage from the masked value. As a traditional higher-order DPA attack would require leakage from both the masked value and the mask, it can not be mounted using the CMOS leakage alone. Thus, a higher-order attack would also have to exploit the leakage from the secure logic style, which should increase its complexity considerably. An alternative avenue of attack which targets the unmasked values in the secure zone, requires the attacker also to overcome the protection of the secure logic style. Thus, the protection offered by the secure logic style can be seen as being rolled over to the complete processor system, even though only a fraction of the system is actually implemented in this logic style.

3. EXTENDED FUNCTIONALITY

The basic concept can be extended with additional features which offer further functions or facilitate the use of the hardware protection mechanism by the compiler and/or the software developer.

3.1 Explicit Association of Masked Values and Masks

In Section 2, the requirement for an addressing scheme for associating masked values and masks has already been mentioned. It is possible to use a pre-existing scheme of the processor or to introduce a new custom scheme. Reusing an already existing addressing scheme might lead to a simpler implementation. Potential candidates are logical and physical register addresses and memory addresses. A disadvantage is that masked values are associated to their masks only indirectly via the storage location of the masked value. Thus, whenever a masked value is moved to a different storage location, e.g. flushed from a register to memory, a mechanism is required to uphold the association with the mask. The compiler is usually in charge of managing the storage location of values, so it could take care of this task.

A more elegant solution is the introduction of a new custom addressing scheme. The processor is thought to have a range of storage locations with unique addresses where each can hold a single masked value and its associated mask. We denote these virtual storage locations as *masked registers* and their addresses accordingly as *masked register addresses*. The masked register address can remain constant and independent of the actual physical location of the masked value and the mask. Such a solution is principally independent of a specific processor architecture and is also much closer to the implementation of a cryptographic algorithm in a high-level language. When cryptographic algorithms operate on (intermediate) values the actual physical location of these values is usually not an issue. Only at compilation time these values are mapped to the storage resources of the processor architecture at hand.

3.2 Alternative Representation of Masks

The security of the protection scheme is based on the condition that masks must never leave the secure zone. As the mask storage unit of the secure zone can only hold a limited number of masks, the number of manageable masked values would also be limited. However, depending on the mechanism for mask generation, it can be possible to find an alternative representation of masks which can leave the secure zone without compromising the overall security. Using this mechanism, the mask storage can be virtually extended or can be shared between different processes in a secure manner.

Finding an alternative representation of masks precludes the direct use of random number generators (RNGs)[2] as mask generators, as their output is not reproducible. However, various kinds of pseudo-random number generators (PRNGs) appear suitable as such mask generators. Seeding of a PRNG must naturally be resistant against reset attacks, where an attacker tries to recreate the system's state by resetting. This can be done for example by seeding the PRNG from an RNG.

The most important characteristics required of the mask generator are the production of at least one fresh mask per clock cycle, ease and efficiency of implementation, sufficient "quality" of the randomness of the sequence of masks, and the capability to reconstruct previous masks via the alternative representation. Regarding the protection against side-channel attacks, it is not necessary to achieve ideal randomness for masks, but it is sufficient to have masks that are uniformly distributed and unpredictable by an attacker [8]. Potential candidates include linear-feedback shift registers (LFSRs) and hardware-efficient stream ciphers, e.g. Trivium [3] or Grain [7].

For our implementation we have chosen a mask generator based on a large maximum-length LFSR, which consequently has a very long period. If the starting point is chosen each time at random (which should be fulfilled by seeding from an RNG), then an attacker should not be able to predict the sequence of produced masks. We stress that it is crucial that the LFSR is seeded from an RNG in order to achieve sufficient quality for the masks. We do not make any claims about the security of the system under the usage of a non-random seed. An attacker could try to use side-channel information from some parts of the LFSR state which circulate outside of the secure zone as part of the alternative representation of masks. We estimate that with the side-channel information available to an attacker it is infeasible to reconstruct the LFSR state or masks. However, we note that this issue remains an open research problem. Furthermore, a maximum-length LFSR will cycle through all possible internal states (except the "dead" state), resulting in a nearly perfect uniform distribution of its output. As alternative representation of a mask we use a given LFSR state and the number of advancements (or steps) of the LFSR from that state which finally produced this mask. We denote this number of steps as the *mask index*. Masks can be stored to memory as a given LFSR state plus the number of steps from that state and the corresponding entry in the mask storage can be reused for a mask with a different masked register address.

[2]Note that RNGs are sometimes also referred to as true/truly random number generators (TRNGs).

If the period of the LFSR is sufficiently long, then masks will not repeat. Thus it is only necessary to use the RNG for seeding of the LFSR at device startup with all masks being subsequently produced by the LFSR. The speed of the RNG thus only dictates some startup cost but has no bearing on the device performance during normal operation. Therefore, a slower but cheaper RNG can be used if the startup cost can be tolerated to be longer. However, it is also possible to build relatively fast and simple RNGs [4] to reduce this initial cost.

Note that any given state of the LFSR can be used as basis for reconstructing masks, as an LFSR can be run in both forward and inverse direction. This property can be used to make the regeneration of masks efficient. Whenever a mask needs to be stored to memory, the current LFSR state can be saved along with it. By the principle of locality, the masks held in the mask storage will not be "far" from the current LFSR state, and thus regenerating them from that state will only take a few steps. In order to uphold security, care must be taken that the words of an LFSR state which is stored to memory are never be used as masks themselves.

All information about the state of the mask generator and mask storage which is held in memory should not be directly readable by an attacker. This is the same requirement imposed on any sensitive data typically stored in memory, e.g. cryptographic keys. Achieving such protection is thus a general issue for any secure embedded system, e.g. by use of a secure operating system preventing dumps of the complete memory, and is out of scope of this work.

It is important to note that instructions which are executed within the secure zone and which have masked operands can only be executed when the corresponding masks are located in the mask storage. If the required masks happen to be stored in memory in their alternative representation, they first need to be restored to the mask storage before regular execution can continue. Flushing out masks from the mask storage can also be used to share the secure zone among multiple processes in a multi-tasking environment. By removing all masks from the mask storage and the clearing of the current state of the mask generator, a complete separation of processes is achievable as it would be required for any environment which runs a secure operating system with potentially insecure user applications. Thus, our solution could be seamlessly integrated into such environments, e.g. processors which support the ARM TrustZone concept [1].

3.3 Exceptional Conditions

The described functionality of the hardware countermeasures can lead to a number of exceptional conditions in the processor. When a new mask needs to be written to the mask storage, there might not be any free entries left (mask storage full exception). Furthermore, the mask generator, which keeps track of the mask index of the generated masks, might encounter an overflow condition of the counter for the mask index (mask index overflow exception). Finally, a mask which is required to execute an instruction with masked operands might not be present in the mask storage, e.g. because it is stored in memory in its alternative representation (mask missing exception). Exceptional conditions should cause hardware traps, so that the according trap handler routines can rectify the situation before normal execution resumes.

4. IMPLEMENTATION DETAILS

We have implemented our protection mechanisms in the 32-bit SPARC V8-compliant Leon-3 processor. Our concrete design decisions are described in the following sections.

4.1 Functional Unit

The functional unit must support all instructions to implement all required cryptographic algorithms. More precisely, all instructions which manipulate critical data, which can potentially be subjected to an SCA attack need to be realized by the protected functional unit. Uncritical operations like updating of round counters, loop condition checking, *etc.* can still be implemented outside of the secure zone using native instructions of the processor. Note that our countermeasures cannot relieve the software developer from avoiding traditional well-known SCA vulnerabilities like data-dependent conditional jumps.

The only class of typical processor operations which can not be protected in our solution are table lookups, as they inherently involve access to the memory. Although data-dependent table lookups are used in some cryptographic implementations, their use should be discouraged as they lead to potential vulnerabilities against cache-based timing attacks [2, 17]. Instruction set extensions can usually be employed to remove the need for table lookups completely, so there is no need to support them in the secure zone concept.

For our implementation we have chosen to include support for AES in the form of the "Advanced Word-Oriented AES Extensions with Implicit ShiftRows" from [14] and a protected XOR instruction. These instructions support all AES round functions and the AES key schedule. Note that as we employ Boolean masking to protect critical values outside of the secure zone, the AES AddRoundKey transformation can also be performed directly on the masked values using an unprotected XOR instruction.

4.2 Mask Generator

For a w-bit processor, the mask generator must be able to deliver a fresh w-bit mask for the result of each instruction with masked operands. In order to sustain an execution rate of one instruction per clock cycle, one fresh mask per cycle is required. Section 3.2 already mentioned our choice of an LFSR-based mask generator. More precisely, we selected a Fibonacci-type LFSR based on the pentanomial $x^{127} + x^{87} + x^{59} + x^{37} + 1$. This irreducible polynomial generates a multiplicative group of order $2^{127} - 1$, which is identical to the period of the LFSR. Thus, after seeding, the LFSR is capable of producing about 2^{122} masks (32 bits each) before the sequence of masks starts repeating.

The choice of the pentanomial has two key advantages. First, the input bit generation can be efficiently parallelized, so that 32-bit pseudo-random masks can be produced in a single clock cycle. Second, the LFSR can also be configured to run in the inverse direction at low extra hardware cost. This functionality is very useful to re-generate masks from their alternative representation in an efficient manner.

The mask generator keeps track of the mask index of the current mask, *i.e.* the number of steps taken from the last seed value. The mask index is written to the mask storage along with the mask and it is an essential part of the alternative representation of the mask. This alternative representation of the mask is used when it needs to be flushed out to the unprotected memory.

4.3 Mask Storage

The mask storage holds a number of masks which can be readily used for any instruction with masked operands. The association of masked values and masks is established via the masked register address (cf. Section 3.1). Each logical register of the processor, *i.e.* in SPARC V8 architectures each of the 32 registers of the current register window, is mapped to a specific masked register address via a custom hardware table (the so-called *masked-register table* or MRT). Moving a masked value from one logical register to another must therefore be reflected by an update of the MRT. Furthermore, the masked register address associated with the mask is contained in the mask storage along with the mask itself. Consequently, the processor can associate masked operands in the logical registers with their respective masks via the masked register address.

We have set the number of entries of the mask storage unit to eight. Each entry consists of a valid bit, a dirty bit, a 10-bit masked register address, the 21-bit mask index, and the 32-bit mask. A mask can be written to any of the eight entries and therefore the organization of the mask storage resembles a fully-associative cache with address-dependent lookup. Mask missing exceptions (cf. Section 3.3) can thus be regarded as the equivalent of cache misses. The valid and dirty bits set automatically by the hardware and can be retrieved via special management instructions. The valid bit indicates whether a specific entry is valid. The dirty bit is set when the mask has no alternative representation in memory[3].

The valid bit, the masked register address, and the mask index of each entry can be read as a single 32-bit word with the help of a custom instruction. Together with the original seed state of the LFSR, this 32-bit word is sufficient to reconstruct the mask and its context at any later point in time. An illustrative example of the interplay of logical registers, the MRT, and the mask storage is provided in Section 4.5.

4.4 Management of Masked Register Addresses

The masked-register table (MRT) is used by the compiler or the developer of the cryptographic application to indicate the presence of specific masked values (with masked register addresses) in the logical registers of the processor. Thus, masked values can be moved freely between logical registers and memory, giving the greatest degree of flexibility to the software. The entries of the MRT can be read and written by custom instructions. As the MRT does not contain any critical data it can be implemented in standard CMOS.

4.5 Masked Values and Masks

In the following, we illustrate the software usage of masked registers with a simple example. An assembly-code example of using the secure zone is given in Figure 3.

Figure 2 shows an example of how masked values are associated to their masks. The standard register file of the Leon-3 with the 32 logical registers is depicted on the left (logical register address and register contents). The MRT with its mapping from logical register addresses to masked register addresses is shown in the middle. For each of the 32 logical registers there is exactly one entry in the MRT. On the right, the mask storage with a capacity of eight entries is shown (only masked register address and mask).

In this example, the register file contains four masked values (A_m, C_m, E_m, and F_m). The mask storage holds five masks (m_A, m_B, m_C, m_D, m_F). The masked register with address 0 encompasses the masked value A_m (held in i0) and the mask m_A (held in the first entry of the mask storage). As both values are resident in the register file and the mask storage, the masked register 0 can be readily used as operand in a secure zone instruction. The same is true for masked register 3 (C_m in i1 and m_C in third entry of mask storage) and for masked register 9 (F_m in o7 and m_F in fifth entry of mask storage). However, the masked register 7, with its masked value E_m in i3, cannot be used, because the associated mask is not present in the mask storage. To become useable, the mask for masked register 7 must first be restored in the mask storage. There are also two masks (m_B and m_D) in the mask storage which are associated with masked registers 2 and 5, respectively. However, the corresponding masked values (B_m and D_m) are not present in the register file. In order to use the masked registers 2 and 5, the masked values must be loaded into a logical register and these registers must be associated with the masked register addresses 2 and 5 by updating the MRT. For example, B_m could be loaded from memory into i2 and the MRT entry for i2 could be updated to point to masked register address 2.

Figure 3 gives an example of implementing a single round of AES on the secure zone in AT&T assembly notation. In this notation, register names are prefixed by % and the destination register of an instruction (if any) is written as the final operand of the instruction. First, the MRT is updated so that the masked register addresses 0-7 are assigned to the logical registers o0-o3 and l4-l7 (`szwwrmrtdir` instruction). Then the AES state in registers o0-o3 is masked (`szmask` instruction). The round key in o4-o7 is applied to the masked AES state via the standard `xor` instruction (cf. Section 4.1). The rest of the round transformations is done via the `szsbox4s` and `szmixcol4s` instructions (cf. [14]) using the temporary registers l4-l7. The result is then unmasked (`szunmask` instruction).

4.6 Integration of the Secure Zone into the Processor Pipeline

Figure 4 gives a simplified view of three pipeline stages of the Leon-3 processor and the logical structure of the secure zone. Each register stage has feedback paths to the previous stages in order to cater for potential data dependencies of subsequent instructions, e.g. if an instructions uses the result of the previous instruction as operand.

While the functional unit with unmasking and masking logic resides on the same level as the execute stage (which contains the other functional units of the processor), the mask storage and mask generator are located on the same level as the register file. This depiction should emphasize, that changes to the mask storage and the mask generator should only occur when there is an according change in the register file. In particular, whenever a masked result of an instruction is committed to the register file, the according new mask must be written to the mask storage.

In some cases, the result of an instruction is calculated in the execute stage, but it is never committed to the register file and thus has no effect on the processor state. This can

[3]If an entry with a set dirty bit is flushed to memory, its alternative representation must be written to memory. If the dirty bit is not set, the entry can just be deleted from the mask storage.

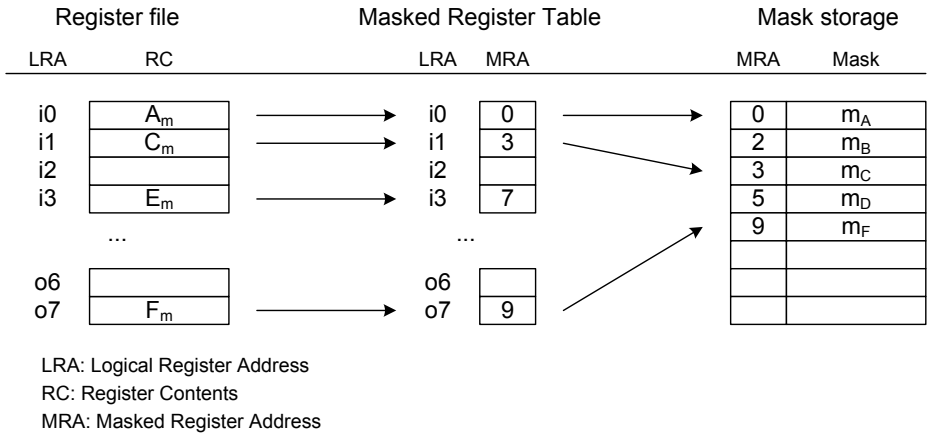


Figure 2: Example of association of masked values and masks.

```
! %o0-%o3: AES state
! %o4-%o7: AES round key
! %l4-%l7: Temporary AES state

! Set up MRT: Assign masked register
! addresses 0-7 to %o0-%o3 and %l4-%l7
szwwrmrtdir %o0, 0
szwwrmrtdir %o1, 1
szwwrmrtdir %o2, 2
szwwrmrtdir %o3, 3
szwwrmrtdir %l4, 4
szwwrmrtdir %l5, 5
szwwrmrtdir %l6, 6
szwwrmrtdir %l7, 7

! Mask AES state
szmask %o0, %o0
szmask %o1, %o1
szmask %o2, %o2
szmask %o3, %o3

! AddRoundKey
xor %o0, %o4, %o0
xor %o1, %o5, %o1
xor %o2, %o6, %o2
xor %o3, %o7, %o3

! SubBytes & 1st part of ShiftRows
szsbox4s %o0, %o1, %l4
szsbox4s %o1, %o2, %l5
szsbox4s %o2, %o3, %l6
szsbox4s %o3, %o0, %l7

! 2nd part of ShiftRows & MixColumns
szmixcol4s %l4, %l6, %o0
szmixcol4s %l5, %l7, %o1
szmixcol4s %l6, %l4, %o2
szmixcol4s %l7, %l5, %o3

! Unmask result
szunmask %o0, %o0
szunmask %o1, %o1
szunmask %o2, %o2
szunmask %o3, %o3
```

Figure 3: A single round of AES implemented on the secure zone.

happen for various reasons, e.g. for instructions after a taken branch or when a previous instruction has caused a trap. Certain control signals of the processor pipeline make sure that such cases are handled correctly. The mask storage and the mask generator can make use of the same control signals in order to determine whether an instruction should change their state or not.

Figure 4 also shows two pipeline register stages within the secure zone which can be regarded as extensions to the pipeline register stages at the end of the execute and memory stage of the processor pipeline. The secure zone must implement feedback paths from these register stages in order to cater for potential data dependencies of subsequent instructions which use the secure zone in the same way as the rest of the processor pipeline.

4.7 Automatic Handling of Exceptional Conditions

Whenever an exceptional condition as described in Section 3.3 occurs, the processor throws a hardware trap and automatically directs execution to the according trap handler. We have added a set of management instructions which allows for an automatic handling of these exceptional conditions[4]. More precisely, an operating system with appropriate trap handlers can manage these conditions transparently for the cryptographic application. Therefore, cryptographic applications need not be aware of the actual size or contents of the mask storage or of the size of the mask index. The only thing it has to do is to indicate where it holds the masked intermediate values during execution via the MRT. These updates of the MRT could be inserted automatically by the compiler, so that a cryptographic application using the protection of the secure zone could be written in almost the same way as an unprotected application.

5. USE IN AN AUTOMATED DESIGN FLOW AND COMPILER INTEGRATION

The presented protection mechanism can be easily used as basis for a fully automated design flow for secure embed-

[4]These instructions allow to read and write the mask generator state, regenerate masks, read and clear entries of the mask storage, update the MRT, *etc.*

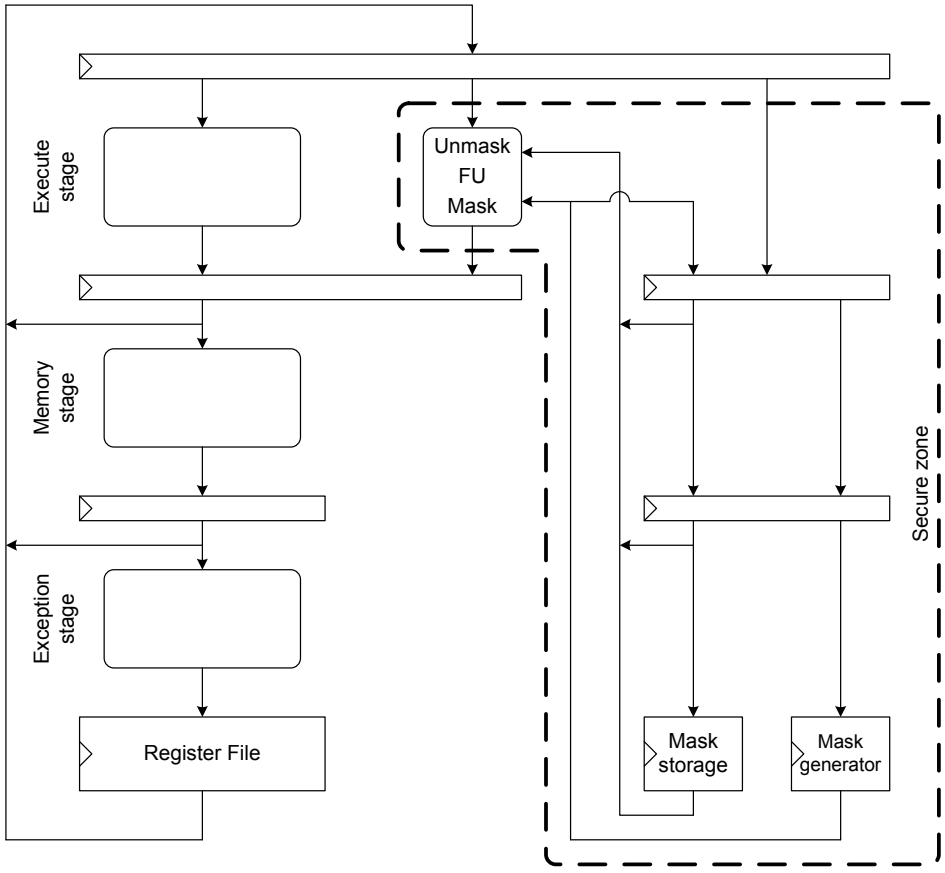


Figure 4: Secure zone components in relation to the processor pipeline stages.

ded systems. Starting from a baseline protected processor, an arbitrary number of protected instructions can be simply added by extending the functional unit within the secure zone with the appropriate capabilities. Moreover, the set of instructions which is required to implement specific cryptographic algorithms could be extracted automatically from a high-level language description of the algorithm, e.g. [10, 11]. At the same time, the design flow can generate accompanying tool-chains and simulators for the customized protected processor which can then be used to develop protected cryptographic software. In the best case, the system designer just needs to decide on the cryptographic algorithms which need to be supported and provide high-level language descriptions of them.

As already mentioned, most of the administrative tasks relating to the use of the secure zone could be offloaded to the compiler. In the ideal case, the only task left to the developer of a cryptographic implementation would be to indicate the mark the key which needs to be protected. From this, the compiler can deduce all intermediate values which depend on the key and which need to be protected by the secure zone. Given the interdependency of these values, the compiler can order the operations in a way which uses the limited number of entries in the mask storage in an optimal way, minimizing the number of times that masks have to be flushed to memory or restored from memory. If masks and masked values are associated via storage location, the compiler could make sure that a masked value is in the correct location when it is operated upon by the secure zone (otherwise the corresponding mask cannot be found). If this association is done via the MRT, the compiler can insert the instructions for updating the MRT automatically.

The compiler can also manage the secure zone directly by keeping track of the number of occupied entries in the mask storage and the current mask index and by inserting calls to flush masks to memory, restore masks from memory, and handle mask index overflows when appropriate. This would require the compiler to model the effects of the program on parts of the secure zone. Managing of the secure zone can also be done at runtime via traps as discussed in Section 4.7.

6. PRACTICAL RESULTS

6.1 Implementation Cost

In order to estimate the hardware overhead introduced by the secure zone, we have performed standard-cell synthesis of two versions targeting the UMC 0.18 µm standard-cell library FSA0A_C from Faraday [5]. The first version of the secure zone (minimal version) includes only the minimal functionality sufficient to protect implementations of AES encryption and decryption (masking only of fixed registers, 7-entry mask storage, mask generator based on 64-bit LFSR, no task switching support, no exception handling). The second version (full version) encompasses all features described in the previous sections (masked register addressing scheme, 8-entry mask storage, mask generator based on

Table 1: Synthesis results for both versions of the secure zone.

Component	Requires Secure Logic	Minimal GEs	Full GEs
Functional unit	yes	2,984	3,303
Mask generator	yes	779	1,994
Mask storage	yes	2,833	7,122
SZ functionality	yes	1,777	4,461
MRT	no	n/a	5,478
Total		8,373	22,358

127-bit LFSR, task switching support, exception handling).

The synthesis results for both variants are given in Table 1. The area requirements for each sub-component are stated separately. Note that the figures refer to implementation in standard CMOS. The column "Secure Logic" indicates whether the given component requires realization in a secure logic style. The component "SZ functionality" encompasses all functions which are not part of the other components, *i.e.* control word decoding, unmasking and masking logic, pipeline register stages, feedback logic, exception detection, and registering of the output word.

The total size of the minimal variant of the secure zone is about 8.4 kGates. Note that the size of the functional unit could be further reduced by employing more light-weight AES instructions [14]. The full variant requires a total of about 22.4 kGates, but the part that needs to be implemented in secure logic (which excluded the MRT) is only twice the size of the minimal variant.

Note that the full implementation includes a number of administrative functions, some of which could be stripped in a practical implementation in order to save area. Furthermore, the exception handling could also be optimized further. Also, with the mechanism for flushing masks to memory, the number of mask storage entries could also be reduced, given a tradeoff between area and execution time.

In any case, the full implementation offers a framework for supporting multiple cryptographic algorithms with little extra cost, as only the required instructions need to be added. For example, the raw cost for AES is only 3 kGates (in CMOS) and could be even reduced to less than 1 kGate [14]. All other components can be reused by the various crytpographic implementations. This is a fundamental difference to cryptographic coprocessors implemented in secure logic, where there is typically one coprocessor per cryptographic algorithm and the overhead for registers and control logic is incurred for each of them. Furthermore, coprocessors are usually not suited to be used by multiple tasks in parallel. Note also that protecting the whole processor system in a secure logic style is currently no viable option to achieve the same degree of functionality offered by our implementation, due to the lack of efficient solutions for adequately protecting external memories.

The overhead incurred by implementation in a secure logic style greatly depends on the characteristics of the chosen logic style. For example, WDDL approximately triples the area [16]. Furthermore, the clock frequency is at least halved and the power consumption is increased by a factor of about 3.5. Note that the decrease in clock frequency could be limited to those execution times where the secure zone is active via frequency scaling.

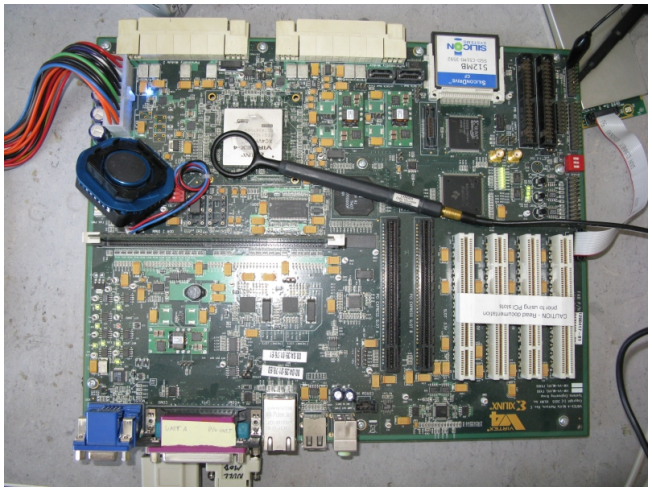


Figure 5: Setup for the DEMA attack.

6.2 Preliminary SCA Evaluation

We have prototyped our implementation on a Xilinx ML410 FPGA board which features a Virtex 4 FX FPGA. Apart from a functional verification in "real" hardware, this enabled us to perform a preliminary evaluation of the SCA resistance of our approach. We compared two implementations of AES: One which uses unprotected AES extensions and one which makes use of the instructions offered by the secure zone. Due to the complexity of using a secure logic style approach on an FPGA and due to time limitations, we had to refrain from implementing the secure zone in a secure logic style. We believe that it is reasonable to assume that a proper implementation of the secure zone in a secure logic style can only lead to a further increase in the practical security. Therefore, our evaluation results can be seen as estimating the lower bound for security. Even though no secure logic style is employed, the protected implementation limits the occurrence of critical values to the secure zone, whereas in the unprotected implementation, critical values are moved through the complete processor pipeline. Hence, the unprotected implementation was expected to be much more susceptible to attacks.

To test this assumption, we performed a DEMA attack with a total of 250,000 power traces on both implementations. The setup is depicted in Figure 5. Despite the relatively high noise in the setup, we were able to successfully attack the unprotected implementation. This attack yielded $\rho \approx 0.02$ which translates to a maximum of about 70,000 required power traces [8]. On the other hand, the attack on the protected implementation did not succeed. The attack has thus been made harder by a factor of at least more than 3.5, just by limiting the circulation of critical values. This factor should multiply to the protection factor offered by the secure logic style. The correlation traces for both attacks are shown in Figure 6 and Figure 7. The correlation is measured between predicted power consumption values under the assumption of a specific value for a byte of the key (key hypothesis) and the actual measured power consumption. If the correlation is significantly higher for a specific key hypothesis in relation to all hypotheses, this indicates that the key value has been guessed correctly and that the attack has been successful. Thus, these correlation traces

show the result of an attack on a particular key byte (the second key byte of the AES cipher key in this case), where the correlation trace for the hypothesis using the correct key byte is in dark gray whereas the correlation traces for the 255 incorrect key hypotheses are displayed in light gray. Note that the attack on the protected implementation has a ghost peak[5] around point 400, but this peak occurs for all key hypotheses.

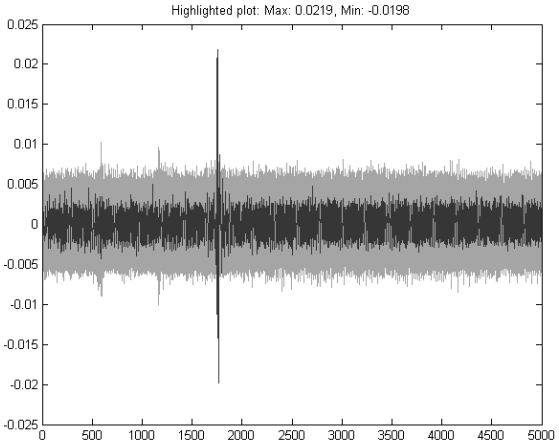


Figure 6: Attack on the unprotected implementation.

6.3 A Note on Higher-Order Attacks

The use of an LFSR as mask generator introduces a linear connection between some of the masks in the sequence. More precisely, a specific bit of a new mask is a linear combination (XOR) of a number of bits of some previous masks (for our choice of LFSR, this are four bits coming from three or four previous masks). So there is a theoretical threat of a higher-order attack using the power consumption connected to these dependent mask bits. As there are at least three masks contributing to any new mask bit, an attacker would need to mount at least a fourth-order attack, which is usually considered impractical. If attacks of this high order should become a threat, the required order can be arbitrarily increased in software by regularly advancing the mask generator and skipping some of the produced masks without using them.

7. CONCLUSIONS

We have presented a detailed concept for protecting embedded processors against SCA attacks. The main contribution of our work is that it takes the roughly sketched ideas from Tillich *et al.* and develops a protection mechanism which builds on state-of-the-art SCA research and incorporates the requirements of modern embedded systems. Our solution makes use of secure logic styles and acknowledges the incurred implementation overhead by requiring only a portion of the processor to be implemented in such a logic style. Requirements of modern embedded systems like multi-tasking and separation into secure operating system

[5]The ghost peak probably relates to the masking operation of the plaintext.

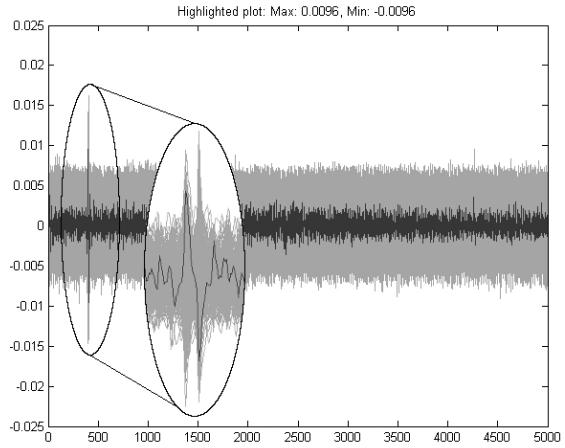


Figure 7: Attack on the protected implementation.

and potentially insecure user applications are catered for. The task of dealing with the hardware countermeasures can be largely offloaded to the operating system and the compiler, so that developers can write protected cryptographic applications in almost the same way as unprotected ones. Our countermeasures are modular, so that certain features can be omitted if they are not needed or if an increased processor workload is acceptable. We have implemented the protection mechanism in two versions for protecting AES on a typical embedded processor, where the minimal version required about 8.4 kGates while the full version could be realized with about 22.4 kGates. Our concepts are applicable to a large range of cryptographic algorithms and are ideally suited for an automated development flow of protected embedded processors.

8. ACKNOWLEDGMENTS

The research described in this paper has been supported by the Austrian ministry BM:VIT in the FIT-IT program line "Trust in IT Systems" under grant 816151 (project POWER-TRUST), and, in part, through the ICT Programme under contract ICT-2007-216676 ECRYPT II. The information in this document reflects only the authors' views, is provided as is, and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

9. REFERENCES

[1] ARM Ltd. TrustZone Technology Overview. `http://www.arm.com/products/security/trustzone/`.

[2] D. J. Bernstein. Cache-timing attacks on AES. Available online at `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`, April 2005.

[3] C. D. Canniére and B. Preneel. TRIVIUM Specifications. eSTREAM, ECRYPT Stream Cipher Project (`http://www.ecrypt.eu.org/stream`), Report 2005/030, April 2005.

[4] M. Dichtl and J. D. Golić. High-Speed True Random Number Generation with Logic Gates Only. In E. Oswald and P. Rohatgi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2008, 10th*

International Workshop, Washington DC, USA, August 10-13, 2008, Proceedings, volume 5154 of *Lecture Notes in Computer Science*, pages 45–62. Springer, August 2008.

[5] Faraday Technology Corporation. Faraday FSA0A_C 0.18 μm ASIC Standard Cell Library, 2004. Details available online at `http://www.faraday-tech.com`.

[6] P. Grabher, J. Großschädl, and D. Page. Non-Deterministic Processors: FPGA-Based Analysis of Area, Performance and Security. In *Proceedings of the 4th Workshop on Embedded Systems Security (WESS 2009)*, pages 1–10. ACM Press, 2009.

[7] M. Hell, T. Johansson, and W. Meier. Grain - A Stream Cipher for Constrained Environments. eSTREAM, ECRYPT Stream Cipher Project (`http://www.ecrypt.eu.org/stream`), Report 2005/010, 2006. Revised version.

[8] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks – Revealing the Secrets of Smart Cards*. Springer, 2007. ISBN 978-0-387-30857-9.

[9] D. May, H. L. Muller, and N. P. Smart. Non-deterministic Processors. In V. Varadharajan and Y. Mu, editors, *Information Security and Privacy, 6th Australasian Conference, ACISP 2001, Sydney, Australia, July 11-13, 2001, Proceedings*, volume 2119 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2001.

[10] L. Pozzi, M. Vuletić, and P. Ienne. Automatic Topology-Based Identification of Instruction-Set Extensions for Embedded Processors. In *Proceedings of the conference on Design, automation and test in Europe (DATE 2002)*, page 1138. IEEE Computer Society, 2002.

[11] S. Ravi, A. Raghunathan, N. Potlapally, and M. Sankaradass. System design methodologies for a wireless security processing platform. In *39th Design Automation Conference, DAC 2002, New Orleans, Louisiana, USA, June 10-14, 2002, Proceedings*, pages 777–782, New York, NY, USA, 2002. ACM Press.

[12] F. Regazzoni, A. Cevrero, F.-X. Standaert, S. Badel, T. Kluter, P. Brisk, Y. Leblebici, and P. Ienne. A Design Flow and Evaluation Framework for DPA-Resistant Instruction Set Extensions. In C. Clavier and K. Gaj, editors, *Cryptographic Hardware and Embedded Systems – CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 205–219. Springer, 2009. ISBN 978-3-642-04137-2.

[13] G. E. Suh, C. W. O'Donnell, and S. Devadas. Aegis: A Single-Chip Secure Processor. *IEEE Design and Test of Computers*, 24(6):570–580, December 2007.

[14] S. Tillich and J. Großschädl. Instruction Set Extensions for Efficient AES Implementation on 32-bit Processors. In L. Goubin and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems – CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pages 270–284. Springer, 2006.

[15] S. Tillich and J. Großschädl. Power-Analysis Resistant AES Implementation with Instruction Set Extensions. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems – CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 303–319. Springer, September 2007.

[16] K. Tiri, D. D. Hwang, A. Hodjat, B.-C. Lai, S. Yang, P. Schaumont, and I. Verbauwhede. Prototype IC with WDDL and Differential Routing - DPA Resistance Assessment. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 354–365. Springer, 2005.

[17] Y. Tsunoo, E. Tsujihara, K. Minematsu, and H. Miyauchi. Cryptanalysis of Block Ciphers Implemented on Computers with Cache. In *International Symposium on Information Theory and Its Applications (ISITA 2002)*, October 2002.

Porscha: Policy Oriented Secure Content Handling in Android

Machigar Ongtang
Faculty of Information Technology
Dhurakijpundit University
Bangkok 10210, Thailand
machigar.ong@dpu.ac.th

Kevin Butler
Department of Computer and Information Science
University of Oregon
Eugene, OR 97403 USA
butler@cs.uoregon.edu

Patrick McDaniel
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA 16802
mcdaniel@cse.psu.edu

ABSTRACT

The penetration of cellular networks worldwide and emergence of smart phones has led to a revolution in mobile content. Users consume diverse content when, for example, exchanging photos, playing games, browsing websites, and viewing multimedia. Current phone platforms provide protections for user privacy, the cellular radio, and the integrity of the OS itself. However, few offer protections to protect the content once it enters the phone. For example, MP3-based MMS or photo content placed on Android smart phones can be extracted and shared with impunity. In this paper, we explore the requirements and enforcement of digital rights management (DRM) policy on smart phones. An analysis of the Android market shows that DRM services should ensure: *a*) protected content is accessible only by authorized phones *b*) content is only accessible by provider-endorsed applications, and *c*) access is regulated by contextual constraints, e.g., used for a limited time, a maximum number of viewings, etc. The *Porscha* system developed in this work places content proxies and reference monitors within the Android middleware to enforce DRM policies embedded in received content. A pilot study controlling content obtained over SMS, MMS, and email illustrates the expressibility and enforcement of Porscha policies. Our experiments demonstrate that Porscha is expressive enough to articulate needed DRM policies and that their enforcement has limited impact on performance.

Categories and Subject Descriptors

C.2 [**Computer-Communication Networks**]: General ; D.4.6 [**Operating Systems**]: Security and Protection—*Access controls, Authentication*

General Terms

Security

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

Keywords

DRM, Mobile Phone Security, Android, Security Policy

1. INTRODUCTION

Mobile phones are used extensively by nearly 5 billion people worldwide [28] and form a vital information conduit for business and personal information access. The use of mobile phones has long transcended strictly voice calling, e.g., SMS exceeded 5 trillion messages worldwide in 2009 [43]. As users perform ever more data-centric activities on their phones, they are increasingly purchasing *smartphones* with the ability to run wide varieties of applications. Over 170 million smartphones were purchased globally in 2009 [18]. The data and applications used on these phones are equally diverse. For example, the Apple App Store, which contains over 130,000 applications, recorded 280 million application downloads in December 2009 [21].

Open platforms such as Android provide few direct protections for the content placed on the phone. Access controls restrict access to application interfaces (e.g., by placing permissions on application components in Android), rather than placing explicit access controls on data they handle. Therefore, what limited content protections exist are largely a by-product of the way interfaces are designed and permissions (often capriciously) assigned. Thus, a malicious application with the appropriate permissions can exfiltrate even the most sensitive of data from the phone. Malware has recently begun to exploit such limitations [3, 4, 5]. Moreover–even in the absence of malicious applications–commercial interests such as media providers wish to provide content without exposing themselves to content piracy.

To combat these issues, a consortium of mobile phone manufacturers including Nokia, LG, Motorola, Samsung, and Ericsson have recently developed standards for content protection on mobile devices. Codified within the Open Mobile Alliance and focusing primarily on pay-per-use content such as ringtones and multi-media, the OMA DRM v1.0 [37] and v2.0 [38] standards define an API and infrastructure for authorizing *devices* to process content. To simplify, OMA DRM devices obtain rights objects (use licenses and cryptographic keys) from providers that allow them to access downloaded content. The licenses can regulate how the content may be used in simple ways such as by discrete lifetime and maximum number of uses. The granularity of the OMA DRM specifications, however, is coarse. The licensing unit is the phone; as a result, the specifications say nothing about content management when it is on the phone. Specifically,

there are no considerations of which applications may access content. This was reasonable when the specification was written in 2004, as there were no application markets at the time and phone manufacturers provided their own software for the phone. Since that time, though, the smartphone revolution has mandated a need for protections at the application layer, now that many applications can be purchased or downloaded to access on-phone content. Otherwise, content is subject to improper use by untrusted applications such as rogue media players.

In this paper, we introduce the Policy ORiented Secure Content Handling for Android (*Porscha*) system. Porscha enforces fine-grained content policies[1] over content delivered to the phone. A study of application markets (see next section) illuminates the needs of providers: from financial transactions and airline tickets, to personal applications such as diaries and journals, the diversity and sensitivity of content processed by smartphones is immense. This study prompts three classes of fine-grained content policies studied throughout: a) content should only be accessible by explicitly authorized phones, b) content should only be accessed by provider endorsed applications, and c) content should be subject to contextual constraints, e.g., used for a limited time, a maximum number of viewings, etc. In supporting these policies, we extend the OMA DRM policy schema [37] to embrace finer-grained controls.

Porscha policies are enforced in two phases: the protection of content as it is delivered to the phone (in transit, see Section 4.2), and the regulation of content use on the phone (on platform, see Section 4.3). For the former, Porscha binds policy and ensures content confidentiality "on the wire" using constructions and infrastructure built on Identity-Based Encryption [11]. For the latter, Porscha enforces policies by proxying content channels (e.g., POP3, IMAP, Active Sync) and placing reference monitor hooks within Android's *Binder* IPC framework. We implement and test Porscha on a T-Mobile G1 smartphone and perform experiments using the three most popular content types: SMS messages, MMS messages, and email. Our experiments with Porscha show that delivery delay for MMS is slightly over 1 second, while latency from processing emails is only about 1 second or less. A security analysis is given and we conclude with a discussion of alternate designs and policy and infrastructure extensions. We begin in the next section by considering whether content policy is necessary for smartphones.

2. DO SMART PHONES NEED DRM?

In looking at DRM in smartphones, we must ask the obvious question of what must the service actually do. To answer this question, we surveyed applications and usage seen in current cell phones to attempt to ascertain what kinds of documents are commonly exchanged and what the reasonable requirements are that the providers may place on them. We evaluated the top 50 free Android applications in each of the 16 application categories present Android Market in April 2010. For the purposes of this initial study, we focused on applications that delivered content via SMS, MMS, or email. Table 1 shows the number of applications requesting permissions to receive SMS and MMS and to read from or write to SMS, MMS, and email attachments. We briefly summarize our findings on content use below:

Personal and Business Documents:.

Applications in the Communication category (e.g. third-party email/SMS/MMS clients), Tools category (e.g. antivirus, backup tools, and Office Viewer), and Travel category (e.g. language translator) in Table 1 frequently manage personal or business SMS, MMS, and emails.

Documents in this category include sensitive emails between business partners and others encompass security capabilities, e.g., SMS is used for authorization in access control systems such as Grey [9]. In these cases, unintended exposure can leak business secrets or compromise the access control system. Thus, providers need to ensure that (a.) only targeted phones (i.e. authorized users) receive the documents, (b.) only trusted client applications can handle them, and that (c.) these documents can never be modified.

Service-specific data:.

A number of applications use SMS to send commands to on-phone clients. Note that in almost all cases, there should only be one legitimate client consumer for the content type– the one provided by the service itself. For instance, one spy camera application used SMS to command the phone capture pictures and record videos. Similarly, Mydroid[2] is a tool for finding the phone by turning off the silent mode and turning up the volume when it receives a command via SMS, and Mobile Defense[3] allows remote connection to the phones after receiving an authenticated SMS message.

Commands in these applications are sent or received via provided websites or other interfaces. Unauthorized exposure of the "command" documents could reveal the application behavior, and indirectly the user's intent. The applications may misbehave if the commands are tampered with. In response, the senders must let (a.) only the phones under control receive the commands, (b.) only the applications to execute the commands to process them, and (c.) the commands read only and may be read only once, and (d.) ensure only legitimate content is consumed by the client.

Financial Information.

Emails and SMS have become key media for financial institutions to communicate with their clients. For instance, banks and credit card companies offer SMS banking, SMS account alerts, and e-statements. Payment service providers such as PayPal and Amazon Payments mainly contact their customers via email and also offer SMS-based payment service. Similar to personal and business documents, the sending institutions aim to inform the users. As a result, the documents must be (a.) sent only to the phones of such particular customers. They must be (b.) accessed only by trusted messaging clients. Moreover, some documents such as payment requests may be designed to work with a group of payment applications trusted by the institutions which can also be identified by their hashes or signatures. In most cases, the senders should also ensure that (c.) these documents are read-only. They should be deleted only through trusted client applications.

[1]Throughout we use the terms DRM and content policies interchangeably. While the latter term is arguably more general, any distinctions are outside the scope of the definition and enforcement of policy studied here.

[2]`http://code.google.com/p/mydroid/`

[3]`https://www.mobiledefense.com/`

Application Category	Receive SMS	Receive MMS	Read SMS	Write SMS	Read Attachment
Communication	7	2	10	6	1
Tools	5	2	6	3	0
Finance	1	0	1	0	0
Travel	1	0	1	0	0
Others	3	1	2	3	1

Table 1: Number of sample applications that access SMS, MMS, and email.

2.1 DRM Policy Requirements

The surprising result of the application analysis was the incredible consistency of the content policy requirements. With few exceptions, the requirements fell into three categories:

- **Binding content to the phone** – most of the applications required that the content be targeted to a single identified user or phone. Failure to implement this policy could have catastrophic consequences (in financial applications), or undermine the entire service (in media access applications).
- **Binding content to endorsed applications** – often observed in desktop environments but largely ignored in the mobile device industry, it is important to control which particular applications can process protected content. The consequences of the failure to enforce this policy are similar to those above—a malicious application on an otherwise legitimate phone could corrupt, exfiltrate, or otherwise misuse delivered content.
- **Constraining continuing use of the content** – it is essential that the provider be able to control not just access, but how that access evolves or expires over time. Frequency, count, or temporal constraints were common. Failure to provide these polices would marginalize the license structures upon which many services are now built.

To illustrate, Table 2 gives example policies falling into these categories (using a self-explanatory policy schema). Policy 1 states that a particular document (for example, an authorization SMS in our access control system example) can be read only if the application is signed with a particular key and the phone is within 50 meters from the place under control (e.g., as used in the Grey system). Policy 2 explicitly identifies a legitimate application by its fingerprint (i.e., a hash of the application image `.apk` file). We revisit these policies in subsequent sections.

3. BACKGROUND AND ASSUMPTIONS

Having considered DRM policy requirements, we now examine how content is currently delivered to mobile phones. We begin this section by briefly describing Android, then discuss content delivery through the network and its handling on the phone itself.

3.1 Android Background

Android is a middleware platform for mobile phones, built on a Linux kernel that isolates applications by having them run in their own process spaces with their own virtual machine instances. Operating system details are hidden from application developers, who access and provide functionality through *components*. These components are assembled to provide applications, which perform all inter-process communication (IPC) through Android's *Binder* mechanism. Components interact primarily through the use of *Intent* messages, which can either explicitly address components by name or through implicit *action strings*, resolved to the appropriate receivers by the Android middleware. Components set up *Intent filters* and specify action strings in order to subscribe to these specific Intents.

There are four types of components in Android. *Activity* components normally provide user interfaces via the touch screen and keypad. *Service* components perform background processing. *Broadcast Receiver* components act as listeners which enable asynchronous event notifications. They usually receive Intents addressed by action strings. Standard action strings include "boot completed" and "SMS received". Lastly, *Content Provider* components act as a persistent data store that implement an SQL interface. If implemented by the providing application, other applications can query, update, and delete the data in the Content Providers. Figure 1 describes how these components interact with each other. In addition, applications have the ability to directly call APIs from other applications.

Applications in Android can be classified into two groups. *System* applications, including the phone, dialer, and messaging applications, are bundled with the phone when it is provisioned to a subscriber and are stored in a read-only system partition. *User* applications are obtained from a variety of sources, including the Android Market, and are installed by users: these programs can compete with or complement system applications, or extend the platform's functionality in very different ways.

Android's security framework is based on *permission labels*, which are unique text strings defined either by applications or the middleware. Application developers specify a list of permission labels in an application's *manifest* file, which presents information about resources an application is allowed to access. For example, receiving and reading an SMS message requires an application to hold the `RECEIVE_SMS` and `READ_SMS` permissions, respectively. In Android, all content is treated equally, meaning that *any application with permissions to access a particular document type can access all documents of that type.* Importantly, elevated protection levels such as allowing "dangerous" functionality for certain permissions rely on the user to confirm all permissions associated with an application at install time; however, users who may not fully understand what they are allowing and consequently may make bad security decisions.

3.2 Documents in Transit

Figure 2(i.) provides an overview of how content is delivered to a phone from outside sources. For clarity, we refer to these external providers of content as *content sources*. Doc-

(1) allow-read{(sig=934*db*3*d*4 . . .) and (location=40.304107,-75.585938,50)} *Only the access control application signed by developer key* `934db3d4` *.... can read the document, and only when the phone is within a 50-meter radius of location (40.304107,-75.585938).*
(2) allow-read{(hash=6*ab*843*a*)} allow-modify{none} allow-delete{(hash=6*ab*843*a*)} *Only the application identified by its binary hash* `6ab843a` *can read and delete the document.*

Table 2: Examples of security policies for content protection.

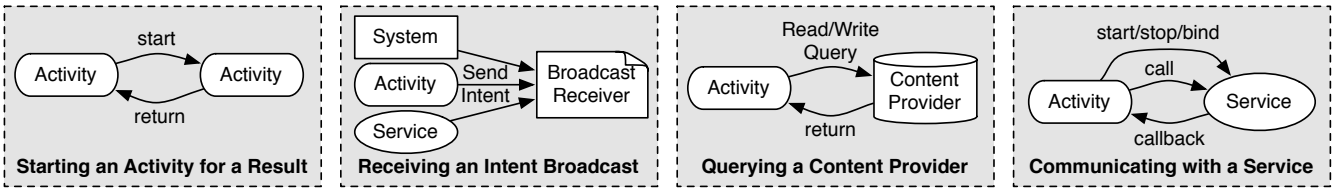


Figure 1: Typical methods of component interaction in Android.

uments sent through the cellular (SS7) network, including SMS and MMS messages, are received at the phone's Radio Interface Layer (RIL), processed by the baseband processor, and made accessible to the phone application. Applications whose content source originates from the Internet, however, connect directly to them in order to receive these documents, such as email. There are no intermediaries on the phone to process this content prior to its handling by the application.

Lack of end-to-end security is a major problem in SMS, MMS, and email transport. SSL/TLS for email delivery only secures connection between the phones and the mail servers. SMS and MMS documents delivered through SS7 make use of security mechanisms found within the cellular network. The heart of the SMS system is the Short Message Service Center (SMSC) which receives short messages from mobile devices inside the cellular network or from external short message entities (e.g. web-based SMS portals). The messages are processed, stored in the SMSC queue, and delivered to the destination devices through a control channel. Messages in transit are encrypted by the network providers.

The MMS system centers on the Multimedia Message Service Center (MMSC). However, phones do not communicate with the MMSC directly but through a WAP gateway push proxy. Upon the arrival of MMS messages, the MMSC notifies the receiving clients with WAP push notifications over SMS. In response, the clients create a TCP/IP connection to retrieve the messages from the MMSC through the WAP gateway push proxy. Clearly, the security of the WAP push notification delivery is based on SMS security. The accompanying MMS message retrieval can be secured using SSL/TLS. More information about the structure of the cellular network and its security is available from Traynor et al. [50].

While SMS and MMS notifications are encrypted, there are still several security issues. GSM encryption is provided only over the radio interface since it is assumed that the SS7 network is inaccessible to external entities. The network providers do not always encrypt SMS messages [20]. Even if they did, though, the employed A5-family encryption algorithms have been compromised: a full rainbow table for the A5/1 cipher has been published [6], while an attack against the KASUMI cipher that is the basis for the new A5/3 cryptosystem can be performed in less than two hours on a PC [14]. Most importantly, GSM encryption does not give end-to-end security because the encryption key is shared between the mobile devices and the network providers, not directly between the content source and receiving device.

3.3 On-Platform Document Access

Access to documents that arrive on the phone is contingent on their method of delivery. Figure 2(ii.) demonstrates how various document types are handled as they arrive at the platform. There are three cases that we consider:

1. **Initial Document Recipients:** These applications either receive documents directly from the platform or from system applications. Their access will be dependent on permission labels set within their manifest files.

2. **Documents at Rest:** Some documents such as SMS, MMS, and the attachments of the emails received will be stored by the phone platform. In Android, these documents will be stored in Content Provider components, which act as databases for this content. Access to these Content Providers to either read or write data is also contingent on permissions in the application's manifest file.

3. **Document Sharing:** *Indirect receivers* are applications receiving documents from other applications. In Android, the APIs allowing interaction and data sharing between applications are mediated by the *Binder* IPC mechanism. Permissions are placed on application components to limit the data that can be sent and received between applications. This acts as a weak method of enforcing information flow enforcement, but is not secure as there is no concept of partial ordering with permission labels such that a lattice may be derived [13].

3.4 Threat and Trust Model

We assume that *the network is untrusted:* an adversary is capable of subverting any communications received from the network interface, regardless of whether the cellular network or the Internet were transited. In addition, any user applications on the phones are assumed to be untrustworthy unless otherwise identified by a sender. Our trusted computing base (TCB) comprises the underlying Linux operating system and the Android middleware itself, as well as system applications.

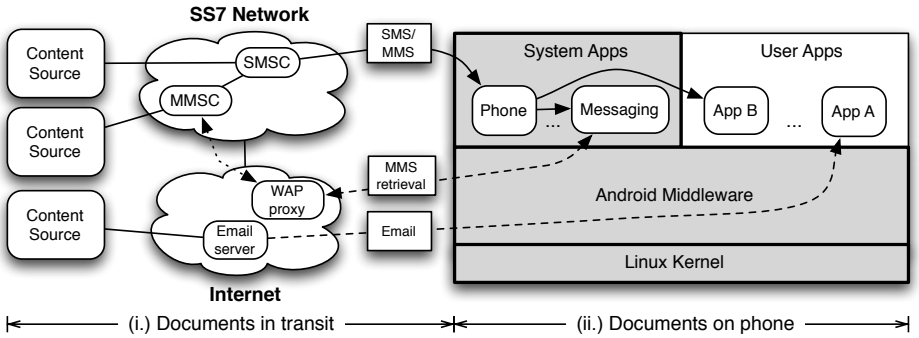


Figure 2: External content providers deliver their documents to the application(s) on the phone. The shaded area represents our TCB.

4. ARCHITECTURE

Porscha enforces security policy in two phases: initially as the content is transmitted over telephony networks and the Internet (as content in transit), and thereafter as it is processed and stored on the phone (on platform). This section describes how Porscha defines and enforces policy in each of these phases.

4.1 Supported Policy

Porscha extends the XML OMA policy [37, 39] to embrace the new DRM requirements identified in our application study, discussed in Section 2.1. All policies are mandatory. That is, any OMA client receiving a Porscha policy must implement the extensions or deny access. Detailed below, policy is encoded in XML as either a section in the text part of MMS or email attachment (see Section 4.3). Porscha supports[4]:

Constraints on Devices – OMA DRM 1.0 binds the content to devices that have acquired the proper license (i.e., right object). OMA DRM 2.0 also supports binding to specific devices identified by the users' International Mobile Subscriber Identity (IMSI) or WAP Identify Module (WIM). Described below, we extend identity to the phone number of the device itself (as regulated by the cellular provider).

Constraints on Applications – Porscha extends OMA DRM to constrain applications consuming content. Here the senders can specify access be restricted to applications with a given code fingerprint (hash of the application image), that are signed with a given developer key, or require that the application be configured with a given set of permissions (this last policy is similar to those found in the Saint system [36]).

Constraints on Use – Common policies codified in OMA DRM, such as validity period and number of uses, are supported. We extend these to support not only the regulation of simple accesses, but also differentiation of simple access from read, modify and delete rights.

4.2 Content-in-Transit

Porscha must secure content delivery over the untrusted networks–namely the *confidentiality*, *integrity*, and *authenticity* of SMS, MMS, and email must be preserved. Porscha uses identity-based encryption (IBE) [11] to ensure these properties. IBE enables the senders to construct the public keys of the recipients from known identities (phone numbers or email addresses), thus eliminating the need for *a priori* key distribution.

We briefly review the structure and use of IBE. Identity-based Encryption systems form a subclass [11] of public key cryptosystems. As with all public key systems, participants (users or other entities) are assigned a public key (which is widely distributed) and a private key (which is kept secret by the participant). What differentiates IBE from other kids of public key systems is the public key itself. An IBE public key is an arbitrary string such as an email address, name, social security number or any other value that is desirable for the target environments. Serving a similar role to a CA in traditional PKI systems, each IBE system contains a trusted private key generator (PKG). The PKG generates private keys using system wide secrets and provides them to the participants through a registration process. The PKG advertises public cryptosystem parameters for the IBE instance.[5] Encryption using the public key is performed by inputting the message (data), public key string, and cryptosystem parameters into the IBE encryption algorithm. Decryption is performed by inputting the ciphertext and private key to the decryption algorithm. Again, as in normal public key systems, it is also possible to encrypt using the private key and decrypt using public key (e.g., as used in creating digital signatures).

We use the following notation below. We denote the private key generator PKG, sender (content source) S, and receiver (phone) R. The identity for participant a is denoted I_a, the public/private key pair for a is K_a^+ and K_a^- respectively, content is m, and a policy for m is p_m. Encryption and signature operations are denoted $E(d, k)$ and $Sign(d, k)$, where d is the input data and k is the key. We denote the one-time time ephemeral key used in the delivery of email as k_e.

SMS/MMS – In SMS/MMS delivery, the recipient's telephone number (MSISDN) is used as their public key identity. Each cellular provider runs a Private Key Generator (PKG) that publishes the IBE public parameters via publicly ver-

[4]For brevity, we omit our XML structure and example polices beyond that identified in Table 1. Note that we are able to encode all policies uncovered in the application study using OMA and the Porscha extensions.

[5]The participant registration process, the secure acquisition process, and other key management services (e.g., revocation) have been discussed at length in other works [19, 31, 8]. Such issues are outside the scope of this work.

ifiable medium, e.g., in a Verisign certificate. The phone's private key and IBE parameters are loaded at subscription time in the phone SIM (see Section 6).

To send SMS/MMS, the sender encrypts its ID, the content, and policy using the receiver's public key. The sender then signs the resulting ciphertext his/her own private key K^-_{sender}. More precisely:

$$S \rightarrow R : E(\{I_S||m||p_m\}, K^+_R)|| \\ Sign(E(\{I_S||m||p_m\}, K^+_R), K^-_S)$$

If the sender uses different PKG than the receiver (e.g., subscribes to a different provider), it contacts the receiver's PKG directly using Internet connection or through multi-domain key management service supported by mobile systems [48, 22, 51]. Note that the addition of the policy, padding, and signature can increase the size of an SMS message beyond that supported by current networks (160 characters). Thus, as described in the following section, we convert SMS messages into MMS messages.

Email – We use a recipient's email address as the target identity. PKGs run in support of email domains publish the public key parameters through extended attributes on DNSsec [27] MX records. Emails can be vary in size and be arbitrarily large. As performing IBE encryption on large content potentially incurs high overhead, the sender encrypts the body of the email using the one-time 128-bit AES symmetric key k_e, which is obtained through Diffie-Hellman key exchange protocol. The symmetric key k_e is in turn encrypted with the receiver's IBE public key. The ciphertext is then signed, and the entirety is sent to the receiver as a MIME encoded email, as follows:

$$S \rightarrow R : E(\{I_S||m||p_m\}, k_e)||E(\{k_e\}, K^+_R)|| \\ Sign(\{E(\{I_S||m||p_m\}, k_e)||E(\{k_e\}, K^+_R)\}, K^-_S)$$

The following section details how these documents are process once they are received by the phone.

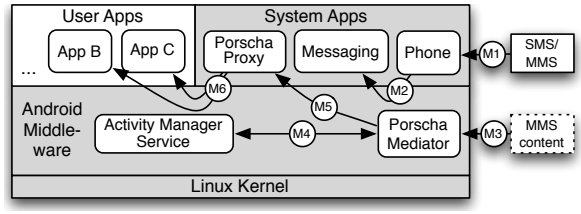
4.3 On-platform Policy Enforcement

When a document arrives at the designated phone, Porscha enforces the content security policy specified by the content source. Detailed in this section, the *Porscha mediator* enforces policy through a set of protocol proxies and authorization hooks within the Android middleware.

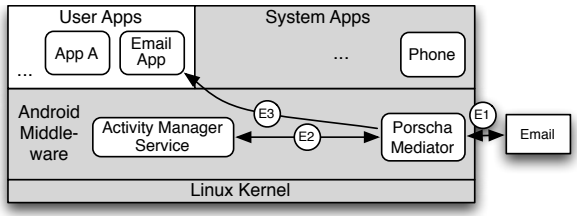
4.3.1 Policy Enforcement on Initial Recipients

SMS/MMS – Depicted in Figure 3(a), SMS and MMS PDUs (the base structure for data messaging services in cellular networks) initially arrive at the Phone application (M1 in Figure 3(a)). Under normal (non-Porscha) circumstances, the Android SMS Dispatcher delivers these messages to all applications that have registered for `WAP_PUSH_RECEIVED` Intents (which requires the `RECEIVE_MMS` permission). The Intent identifies the URI to the MMS content. Messaging application then downloads the MMS content and stores it in the MMS Provider inside of the middleware. The MMS Provider broadcasts another Intent when the content is stored.

To enforce policy, Porscha delays the initial `WAP_PUSH_RECIEVED` broadcast, but automatically triggers the Messaging application to download the content (M2). Once the content download completes, a second mediator hook parses



(a) Mediating SMS/MMS Delivery



(b) Mediating Email Delivery

Figure 3: The Porscha mediator implemented inside Android intercepts the document delivery and enforces the policy on initial recipients.

the PDU (M3), extracts the policy from the content, and determines, in conjunction with the Activity Manager Service (M4), which applications are allowed to receive the document. Applications compliant with the policy receive the subsequent notifications, while those not in compliance will not.

Note that the Intent notifying the arrival of SMS/MMS cannot be issued from within the Android middleware: the broadcasting entity must be an application that possesses `BROADCAST_SMS` permission for SMS and `BROADCAST_WAP_PUSH` for MMS. Additionally, applications must be signed by the platform key for these permissions to be granted. To address this problem, we implement the *Porscha Proxy* as a system application signed and built with the platform. It receives from the Porscha mediator the list of the applications allowed to receive the document, the document content, the document metadata, and whether the document should be dispatched as SMS or MMS (M5). If the document should be dispatched as SMS, the Porscha Proxy constructs an SMS PDU and broadcasts an `SMS_RECEIVED` Intent to authorized applications (M6). If the document is dispatched as MMS, the proxy broadcasts a `WAP_PUSH_RECEIVED` Intent, containing the URI to the MMS content which is accessible through the MMS Content Provider, to authorized applications.

Email – Email traffic is opaque to Android: email client applications use application level protocols such as POP [25], IMAP [26], or Active Sync [33] to communicate with remote mail servers. For this reason, Porscha must "shim" email traffic by creating transparent proxies. The email enforcement intercepts email traffic at the network level through an SSL socket (E1). This mechanism operates at the middleware level inside of our TCB which is inaccessible to the applications. Messages are intercepted and interpreted within each proxy and policy enforced. We use the Apache Mime4j library [7] to parse the e-mail message streams in plain RFC-882 and MIME formats. For each email, the XML attachments are examined. If the attachment is recognized as content policy, the Porscha mediator coordinates with Activity Manager Service to enforce the policy (E2). The content may only be retrieved by an email client if it satisfies the

policy (E3).

For usability, rather that filter email from applications that fail policy, we chose to mask its content. In such cases, Porscha removes all information from the email's header and body, and replaces these fields with the string *Hidden*. We will extend policy schema in the future to allow the sender to provide "alternate text" that would instruct the user where to go to obtain an appropriate application or license for the received content in the event the accessing phone/application does not satisfy the policy.

Note that while controlling access to DRM-protected documents, Porscha allows unprotected documents to be received without restriction. These documents are not IBE encrypted by Porscha. Thus, they can be accessed by the receiving applications.

4.3.2 Policy Enforcement on Documents at Rest

By default, Android stores the SMS, MMS, and email attachments with the system applications using Content Provider components. Applications with permissions to access (read or write) these Content Providers can access this content even if they are not the initial recipients. To allow external senders to control access to the documents delivered from them, we add an extra *policy* field to the structure of each Content Provider record. The Porscha mediator inserts the policy (if available) into this field, and when the Content Provider record is accessed the corresponding policy is checked, and access allowed or denied based on the compliance of the caller application with the policy.

4.3.3 Enforcement on Indirect Receivers

Porscha mediates passing of data between applications as shown in Figure 4.3.3 and as follows:

Intent – The Porscha mediator acts as a reference monitor for Intents that pass protected content. The sending application binds the policy with the Intent that encompasses the content. The mediator prevents applications not satisfying the policy from receiving the Intent.

Content Sharing – Inclusion of a *policy* field into Content Provider records, as described above, allows the Porscha mediator to ensure that every access to stored content satisfies the attached policy. Access is mediated through the Content Resolver mediation hook.

Inter-application API calls – When an application API is called by another application (e.g. Service call), we bind a policy to the input parameter or return value containing content delivered in an Android's *parcel* object. The mediator interprets the parcel, and enforces the policy. If the policy fails (on either the call or return), a security exception is thrown.

5. EVALUATION

This section briefly evaluates the costs of policy enforcement in Porscha. All experiments were executed on a HTC G1 Dream smartphone over T-Mobile 3G services. Porscha was built on the Cyanogen [1] Android 2.1 firmware build and installed on the phone, as was the Stanford IBE library V.0.7.2 (a C implementation of Boneh-Franklin IBE [11]). The IBE module was crossed compiled for the ARM processor as a native executable. Each experiment was repeated 10 times and the average reported (with negligible observed sample variance).

Highlighted in Table 3, an initial set of experiments sought to measure the overheads associated with SMS processing. Here we measured the time between the arrival of the PDU and the time it is dispatched to consuming applications. The experiments showed that SMS processing time is less than 0.1 seconds in unmodified Android. SMS documents are delivered as MMS introducing an additional 4 or greater second overhead. Microbenchmarking of SMS processing revealed three central underlying costs: MMS push notification handling ($\approx$ 1.03 s), MMS content retrieval ($\approx$ 1.44 s), and other connection management processing ($\approx$ 1.04 s). The lower costs associated with SMS-over-MMS vs. MMS with media content were associated with the reduced size of the objects being downloaded (SMS policy objects were on the order of 100s of bytes versus 18kb `.jpg` objects in MMS experiments). The maximum observed overhead for IBE was about 480 msec.

For MMS, we measure the latency from the arrival of the PDU to the time to MMS content is completely downloaded and applications notified. Without IBE, Porscha incurs about a 4% overhead ($\approx$ 20 msec). IBE adds about 1 s. to the overhead–significantly more than in SMS. Here again the cause is the size of the encrypted media: the `.jpg` object. Note that recent advances in IBE offer run-time improvements that can reduce these overheads by as much as 20-35% [23], and techniques such as the use of one time symmetric keys (as detailed in section 4.2) may substantially reduce these costs.

The overhead of processing email ranges from 0.7 seconds to just over 1 second, depending on the email access protocols. These costs are largely due to the proxying of the access protocols, SSL, buffering, email message reconstruction, and policy extraction and evaluation (if policy is available). Note that we have not yet implemented IBE for email, but the experiments above suggest that overheads will be manageable.

6. DISCUSSION

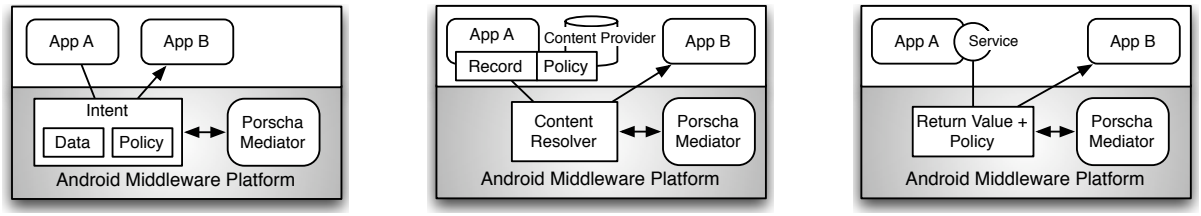
This section examines the security guarantees provided by Porscha and potential threats to these guarantees.

6.1 Protecting the Private Key

Porscha's model for key distribution involves the client phone receiving a private key from the phone provider. Recall from Section 4.2 that a private key generator (PKG) is trusted to create IBE keys and that a phone's MSISDN can be used as a public key identity. The cellular provider will operate the PKG and provide the private key at subscription time on the client's SIM card. However, the SIM is merely a memory card, and is susceptible to being stolen or lost. Therefore, we use a shared secret between the provider and client phone to encrypt the stored private key, with knowledge of the secret required to unlock the key package. One way of communicating this to the user would be with on a slip of paper or a similar out-of-band method when the SIM is purchased or reprogrammed. This method is already used for the SIM authentication key, stored by the provider in their authentication center (AUC).

6.2 Recipients Without Porscha

External senders do not have prior knowledge about whether Porscha is available on client phones. In the absence of



(a) Mediate Intent Passing

(b) Mediate Content Provider Access

(c) Mediate Service Call

Figure 4: Porscha Mediator intercepts cross-application content passing

	Original Latency	Overhead from Porscha without IBE		Overhead from Porscha with IBE	
		policy passed	policy failed	policy passed	policy failed
SMS	0.083	4.07	4.12	4.57	4.56
MMS	5.16	0.22	0.21	1.52	1.53
POP3	6.34	0.68	0.91	2.51	2.61
IMAP	3.79	1.02	1.02	2.94	2.85
Active Sync	3.38	0.8	0.85	2.18	2.19

Table 3: Overheads in processing SMS, MMS, and Email (in seconds).

Porscha, the clients would not be able to access protected documents. This is reasonable, as any content delivered to and intended for the phone should remain opaque to the user. As a result, whether it be phones that do not have a Porscha framework installed or another means of accessing content, e.g., retrieving emails on a computer, content protected by Porscha should and will be inaccessible by these entities.

Note that emails accessed by the IMAP protocol are ultimately managed by an IMAP server; thus, any modifications made to an email by Porscha may be reflected on other clients. To resolve this issue, we store all modifications, such as decrypted emails and those with information removed, locally on the phone, and only reflect back to the IMAP server the original email that was sent to the phone - thus, an original copy of the email is always maintained.

6.3 Application and Platform Trust

With Porscha, we are making assumptions of trust in the Android middleware and associated system applications. There are a number of reasons why this level of trust can be considered appropriate. First, Android applications are signed with a certificate whose private key is held by the system developer; this provides a means of ensuring that the application's integrity is intact and that the origin of the code is as expected. Tools such as Kirin [16] allow install-time certification of applications against potentially dangerous functionality.

Android is middleware that runs on a Linux kernel. Several methods of ensuring kernel integrity have been considered, and this is an area of ongoing research. These include run-time monitors such as the Linux kernel integrity monitor (LKIM) [32]. Ensuring that the phone platform itself is booted into a trustworthy state has also been an area of considerable focus. One promising solution is to include trusted platform modules (TPMs) [49] inside mobile phones; specifically, the *Mobile Trusted Module* (MTM) initiative [15] has considered a TPM-like device that adds functionality for secure boot, which enforces integrity protection of the underlying firmware and system state, and allows for continual assurance of boot-time guarantees through use of the Linux Integrity Measurement Architecture (IMA) [45].

6.4 Alternative Application Enforcement Infrastructures

The extent to which Porscha protects a document largely depends on the attached policies. The senders can indicate the target applications by different degrees of specification from unique application package hashes to loosely defined application properties.

An even stronger security model can be implemented as a performance trade-off. For example, Porscha can be used along with Saint [36] which regulates application interactions (but does not examine the content being passed). As a result, we would gain a more comprehensive view of application behavior and could ensure that all applications are monitored for sharing violations.

Adoption of more heavyweight mechanisms offering continual content monitoring, such as dynamic taint analysis [12, 52], is also possible. However, these systems are not designed for information with semantically rich policy attached. The policies for incoming documents are mostly unique. Managing large and highly dynamic set of taint markings (e.g. in taint analysis) can thus be burdensome. Porscha's content enforcement mechanism is comparatively quite lightweight.

6.5 Digital Rights Management

DRM has been contentiously discussed for nearly 15 years. Such controversy stems from the primary application of DRM: to restrict the use of digital content and prevent piracy, ostensibly to preserve artistic integrity and protect revenue streams for content creators. For example, three competing DRM technologies for mobile or portable devices are Apple's FairPlay [29], Microsoft PlayReady [34]. Along with OMA DRM, all aim to limit media usage for commercial purposes. In Android, OMA DRM 1.0 is supported to manage ringtones, MMS, and pictures, preventing users for forwarding these documents. An external generic framework for DRM implementation is also available but is not used by the official platform. An external framework containing the Open-

Core Content Policy Manager (CPM) [40] is also available. CPM does not implement any DRM algorithms or protocols, but acts as an aggregator with interfaces for authentication, authorization, and access control. Plugins are available for multiple DRM agents, such as WMDRM [42] and DivX [41].

There has been significant opposition to DRM [17, 2] with detractors viewing it as a means for limiting consumer rights and eliminating "fair use" provisions. However, DRM is by definition a generic term for access control technology that secures content and limits its distribution [10, 24]. We consider DRM's role in Porscha strictly as a means to providing content-based access control without comment on business, legal, or philosophical issues. We differentiate from existing DRM schemes, however, and provide a superset of functionality by preserving confidentiality, integrity, and availability, not merely employing encryption and licensing as with typical DRM implementations. In addition, Porscha is lightweight and designed with mobile solutions in mind; by contrast, many advanced DRM protocols are heavyweight and not transparent to applications.

7. RELATED WORK

Mobile phone security often involves regulating the behavior of individual applications installed on the phone to protect the platform. As permissions requested by Android applications reflect their capabilities, Kirin [16] prevents installation of malware by identifying potentially dangerous applications based on these permissions. By contrast, Saint [36] modifies the Android middleware to enforce application policies which regulate how application permissions are granted and how applications interact with each other. While Saint concentrates on securing communication endpoints, Porscha concentrates on the actual content passed.

Enhancing mobile phone security through mandatory access control (MAC) and trusted hardware, specifically Security-Enhanced Linux (SELinux) [47] and TPMs, is a means of protecting application and platform integrity. SELinux security policy has been applied to ensure the integrity of the Openmoko phone platform and trusted applications [35]. Additionally, Rao and Jaeger [44] developed an SELinux-based MAC system that considers input from multiple stakeholders to develop policies for controlling application permissions. Recently, Shabtai et al. [46] have ported SELinux to Android and enabled security policy for enhancing the protection of system processes. Unlike Porscha, which enforces MAC policies to secure documents arriving at the phones, these approaches focus strictly on platform security; while they are orthogonal to our concerns, platform trustworthiness will increase the security of all overlying layers above, including Porscha.

Several IBE solutions have been proposed for use with mobile phones. Mobile phone numbers are commonly used as client identities because they can be effortlessly authenticated by the network. Communication among different network providers running their own PKGs is a major challenge for IBE implementation; proposed solutions have included the use of hierarchical IBE [22, 51] and cross-domain key extensions [48] .

8. CONCLUSION

This paper has proposed Porscha, a content protection framework for Android that enables content sources to express security policies to ensure that documents are sent to targeted phones, processed by endorsed applications, and handled in intended ways. Through a study of real-world applications, we formed an initial scope of appropriate content policies, and we demonstrated how these may be used in Porscha to protect SMS, MMS, and email documents. Porscha secures content delivery using identity-based encryption and mediates on-platform content handling to ensure conformance with content policy. Future work will examine additional types of content that may be protected by Porscha and the policy implications of managing this content.

9. REFERENCES

[1] Android Community ROM. `http://www.cyanogenmod.com/`, March 2010.

[2] I hate DRM: A site dedicated to reclaiming consumer digital rights. `http://ihatedrm.com`, June 2010.

[3] Mobile Watchdog. `http://www.mymobilewatchdog.com/`, January 2010.

[4] SMS Trap. `http://www.smstrap.com/`, January 2010.

[5] Stealth SMS. `http://stealthsms.trusters.com/s_features.htm`, January 2010.

[6] A5/1 Security Project. Creating A5/1 Rainbow Tables. `http://reflextor.com/trac/a51`, 2009.

[7] Apache Software Foundation. Apache James Mime4j. `http://james.apache.org/mime4j/`, March 2010.

[8] G. Appenzeller, L. Martin, and M. Schertler. Identity-Based Encryption Architecture and Supporting Data Structures, Jan. 2009. IETF RFC 5408.

[9] L. Bauer, S. Garriss, J. M. Mccune, M. K. Reiter, J. Rouse, and P. Rutenbar. Device-enabled authorization in the grey system. In *Proceedings of the 8th Information Security Conference (ISC'05)*, pages 431–445, 2005.

[10] E. Becker, W. Buhse, D. Günnewig, and N. Rump, editors. *Digital Rights Management Technological, Economic, Legal and Political Aspects*. Springer, 1 edition, 2003.

[11] D. Boneh and M. Franklin. Identity-Based Encryption from the Weil Pairing. In *Proceedings of CRYPTO*, 2001.

[12] J. Clause, W. Li, and A. Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA)*, pages 196–206, 2007.

[13] D. E. Denning. A Lattice Model of Secure Information Flow. *Commun. ACM*, 19(5):236–243, May 1976.

[14] O. Dunkelman, N. Keller, and A. Shamir. A Practical-Time Attack on the A5/3 Cryptosystem Used in Third Generation GSM Telephony. In *Proceedings of the 30th Annual Cryptology Conference (CRYPTO 2010)*, 2010.

[15] J.-E. Ekberg and M. Kyläanpää. Mobile Trusted Module (MTM) - An Introduction. Technical Report NRC-TR-2007-015, Nokia Research Center, Helsinki, Finland, Nov. 2007.

[16] W. Enck, M. Ongtang, and P. McDaniel. On Lightweight Mobile Phone Application Certification. In *Proceedings of ACM CCS*, November 2009.

[17] Free Software Foundation, Inc. The Campaign to Eliminate DRM. http://www.defectivebydesign.org/, June 2010.

[18] Gartner. Gartner Says Worldwide Mobile Phone Sales to End Users Grew 8 Per Cent in Fourth Quarter 2009; Market Remained Flat in 2009. http://www.gartner.com/it/page.jsp?id=1306513, Feb. 2010.

[19] C. Gentry. Certificate-Based Encryption and the Certificate-Revocation Problem. *Advances in Cryptology*, 2656, January 2003.

[20] M. Gholami, S. M. Hashemi, and M. Teshnelab. A Framework for Secure Message Transmission Using SMS-Based VPN. *Research and Practical Issues of Enterprise Information Systems II*, 1:503–511, 2008.

[21] GigaOm. The Apple App Store Economy. http://gigaom.com/2010/01/12/the-apple-app-store-economy, Jan. 2010.

[22] J. Horwitz and B. Lynn. Toward Hierarchical Identity-Based Encryption. In *Proceedings of EUROCRYPT '02*, pages 466–481, London, UK, 2002. Springer-Verlag.

[23] J.-S. Hwu, R.-J. Chen, and Y.-B. Lin. An Efficient Identity-Based Cryptosystem for End-to-End Mobile Security. *IEEE Trans. Wireless Comm.*, 5(9):2586–2593, September 2006.

[24] R. Iannella. Digital Rights Management (DRM) Architectures. *D-Lib Magazine*, 7(6), 2001.

[25] IETF Network Working Group. Post Office Protocol - Version 3. http://www.ietf.org/rfc/rfc1939.txt, May 1996.

[26] IETF Network Working Group. Internet Message Access Protocol - Version 4, rev1. http://www.ietf.org/rfc/rfc1939.txt, March 2003.

[27] IETF Network Working Group. DNS Security Introduction and Requirements. http://www.ietf.org/rfc/rfc4033.txt, March 2005.

[28] ITU. Measuring the Information Society. http://www.itu.int/ITU-D/ict/publications/idi/2010/index.html, 2010.

[29] S. Jobs. Thoughts on Music. http://www.apple.com/hotnews/thoughtsonmusic/, February 2007.

[30] M. Kirkpatrick and E. Bertino. Enforcing Spatial Constraints for Mobile RBAC Systems. In *Proceedings of the 15th ACM symposium on Access control models and technologies*, 2010.

[31] B. Lee, C. Boyd, E. Dawson, K. Kim, J. Yang, and S. Yoo. Secure Key Issuing in ID-based Cryptography. In *Proceedings of the ACSW Frontiers Workshop*, 2004.

[32] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, and C. D. McDonell. Linux Kernel Integrity Measurement Using Contextual Inspection. In *Proceedings of ACM STC*, 2007.

[33] Microsoft Corporation. ActiveSync HTTP Protocol Specification, version 6.0. http://msdn.microsoft.com/en-us/library/dd299446(EXCHG.80).aspx, May 2010.

[34] Microsoft Corporation. Microsoft PlayReady. http://www.microsoft.com/playready/default.mspx, June 2010.

[35] D. Muthukumaran, A. Sawani, J. Schiffman, B. M. Jung, and T. Jaeger. Measuring Integrity on Mobile Phone Systems. In *Proceedings of ACM SACMAT*, June 2008.

[36] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically Rich Application-Centric Security in Android. In *Proceedings of Annual Computer Security Applications Conference (ACSAC)*, December 2009.

[37] Open Mobile Alliance Ltd. Rights Expression Language Version 1.0. Technical Report OMA-Download-DRMREL-V1_0-20040615-A, Open Mobile Alliance, June 2004.

[38] Open Mobile Alliance Ltd. DRM Architecture 2.0.1. Technical Report OMA-AD-DRM-V2_0_1-20080226-A, Open Mobile Alliance, February 2008.

[39] Open Mobile Alliance Ltd. DRM Rights Expression Language Version 2.0.2. Technical Report OMA-TS-DRM_REL-V2_0_2-20080723-A, Open Mobile Alliance, July 2008.

[40] PacketVideo Corporation. Content Policy Manager Developer's Guide OHA 1.0 r.1. November 2008.

[41] PacketVideo Corporation. PV Android DivX Premium Package. July 2009.

[42] PacketVideo Corporation. PV Android Windows Media Package. November 2009.

[43] Portio Research. Mobile Messaging Futures 2010-2014: Analysis and Growth Forecsts for Mobile Messaging Markets Worldwide, 2010.

[44] V. Rao and T. Jaeger. Dynamic Mandatory Access Control for Multiple Stakeholders. In *Proceedings of ACM SACMAT*, June 2009.

[45] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of the 13th USENIX Security Symposium*, Aug. 2004.

[46] A. Shabtai, Y. Fledel, and Y. Elovici. Securing Android-Powered Mobile Devices Using SELinux. *IEEE Security and Privacy*, 8:36–44, 2010.

[47] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux Security Module. Technical Report 01-043, NAI Labs, 2001.

[48] M. Smith, C. Schridde, B. Agel, and B. Freisleben. Securing Mobile Phone Calls with Identity-Based Cryptography. *LNCS: Advances in Information Security and Assurance*, 5576:210–222, June 2009.

[49] TCG. *TPM Main: Part 1 - Design Principles*. Specification Version 1.2, Level 2 Revision 103. 2007.

[50] P. Traynor, P. McDaniel, and T. La Porta. *Security for Telecommunications Networks*. Advances in Information Security. Springer, July 2008.

[51] Z. Wan, K. Ren, and B. Preneel. A Secure Privacy-Preserving Roaming Protocol Based on Hierarchical Identity-Based Encryption for Mobile Networks. In *Proceedings of ACM WiSec*, 2008.

[52] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing System-Wide Information Flow for Malware Detection and Analysis. In *Proceedings of ACM CCS*, 2007.

Kells: A Protection Framework for Portable Data *

Kevin R.B. Butler
Department of Computer & Information Science
University of Oregon, Eugene, OR
butler@cs.uoregon.edu

Stephen E. McLaughlin and
Patrick D. McDaniel
Systems and Internet Infrastructure Security
Laboratory (SIIS)
Penn State University, University Park, PA
{smclaugh,mcdaniel}@cse.psu.edu

ABSTRACT

Portable storage devices, such as key-chain USB devices, are ubiquitous. These devices are often used with impunity, with users repeatedly using the same storage device in open computer laboratories, Internet cafes, and on office and home computers. Consequently, they are the target of malware that exploit the data present or use them as a means to propagate malicious software.This paper presents the Kells mobile storage system. Kells limits untrusted or unknown systems from accessing sensitive data by continuously validating the accessing host's integrity state. We explore the design and operation of Kells, and implement a proof-of-concept USB 2.0 storage device on experimental hardware. Our analysis of Kells is twofold. We first prove the security of device operation (within a freshness security parameter Δ_t) using the LS^2 logic of secure systems. Second, we empirically evaluate the performance of Kells. These experiments indicate nominal overheads associated with host validation , showing a worst case throughput overhead of 1.22% for read operations and 2.78% for writes.

1. INTRODUCTION

Recent advances in materials and memory systems have irreversibly changed the storage landscape. Small form factor portable storage devices housing previously unimaginable capacities are now commonplace today–supporting sizes up to a quarter of a terabyte [15]. Such devices change how we store our data; single keychain devices can simultaneously hold decades of personal email, millions of documents, thousands of songs, and many virtual machine images. These devices are convenient, as we can carry the artifacts of our digital lives wherever we go.

The casual use of mobile storage has a darker implication. Users plugging their storage devices into untrusted hosts are subject to data loss [16] or corruption. Compromised hosts have unfettered access to the storage plugged into their interfaces, and therefore have free rein to extract or modify its contents. Users face this risk when accessing a friend's computer, using a hotel's business office, in university computer laboratories, or in Internet cafes. The risks here are real. Much like the floppy disk-borne viruses in the 1980's and 90's, malware like Conficker [22] and Agent.bz [32] exploit mobile storage to propagate malicious code. The compromise of hosts throughout military networks, due to malware propagated by rogue portable storage devices, has already led to a ban of their use by the US Department of Defense [28]. The underlying security problem is age-old: users cannot ascertain how secure the computer they are using to access their data is. As a result, all of their information is potentially at risk if the system is compromised. This paper attempts to address this conundrum by responding to the following challenge: *How can we verify that the computer we are attaching our portable storage to is safe to use?*

Storage security has recently become an active area of investigation. Solutions such as full-disk encryption [25] and Microsoft's BitLocker to Go [18] require that the user supply a secret to access stored data. This addresses the problem of device loss or theft, but does not aid the user when the host to which it is to be attached is itself untrustworthy. Conversely, BitLocker (for fixed disks) uses a trusted platform module (TPM) [36] to seal a disk partition to the integrity state of the host, thereby ensuring that the data is safeguarded from compromised hosts. This is not viable for mobile storage, as data is bound to the single physical host. In another effort, the Trusted Computing Group (TCG) has considered methods of authenticating storage to the host through the Opal protocol [37] such as pre-boot authentication and range encryption and locking for access control. These services may act in a complementary manner to our solution for protecting mobile storage from potentially compromised hosts.

In this paper, we introduce *Kells*[1], an intelligent USB storage device that validates host integrity prior to allowing read/write access to its contents, and thereafter only if the host can provide ongoing evidence of its integrity state. When initially plugged into an untrusted device, Kells performs a series of *attestations* with trusted hardware on the host, repeated periodically to ensure that the host's integrity state remains good. Kells uses integrity measurement to ascertain the state of the system and the software running on it at boot time in order to determine whether it presents a safe platform for exposing data. If the platform is deemed to be trustworthy then a trusted storage partition will be exposed to the user; otherwise, depending on a configurable policy, the device will either mount only a "public" partition with untrusted files exposed or will not mount at all. If at any time the device cannot determine the host's integrity state or the state becomes undesirable, the protected partition becomes inaccessible. Kells can thus ensure the integrity of data on a trusted storage partition by ensuring that data can only be written to it from high-integrity, uncompromised systems. Our

*This work was supported by Symantec Research Labs and by grant NSF CCF-HECURA 0937344.

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

[1]The Book of Kells is traceable to the 12^{th} century Abbey of Kells, Ireland due to information on land charters written into it.

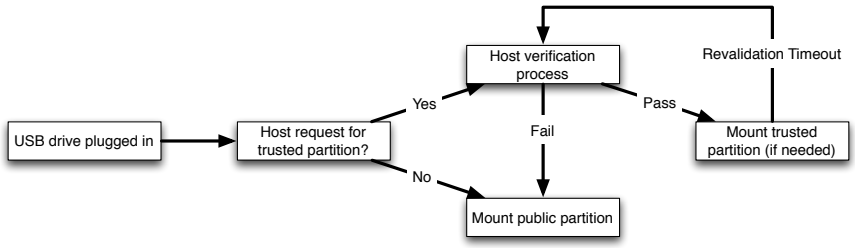


Figure 1: Overview of the Kells system operation. Attestations of system state are required to be received successfully by the device in order for the integrity of the host to be proved, a necessary precondition for allowing data to be available to the host.

design uses the commodity Trusted Platform Module (TPM) found in the majority of modern computers as our source for trusted hardware, and our implementation and analysis use it as a component. *We note, however, that it is not integral to the design: any host integrity measurement solution (e.g., Pioneer [26]) can be used.*

Kells diverges substantially from past attempts at securing fixed and mobile storage. In using the mobile storage device as an autonomous trusted computing base (TCB), we extend the notion of self-protecting storage [3, 11, 21] to encompass a system that actively vets the devices that make use of it. A Kells device is active in order to be able to make these policy decisions. While this is a change from the passive USB devices often currently employed, we note that an increasing class of storage devices include processing elements such as cryptographic ASICs. We thus provide a path to enjoying the convenience of now-ubiquitous portable storage in a safe manner. Our contributions are as follows:

- We identify system designs and protocols that support portable storage device validation of an untrusted host's initial and ongoing integrity state. To the best of our knowledge, this is the first use of such a system by a dedicated portable storage device.
- We reason about the security properties of Kells using the LS^2 logic [6], and prove that the storage can only be accessed by hosts whose integrity state is valid (within a security parameter Δ_t).
- We describe and benchmark our proof of concept Kells system built on a DevKit 8000 board running embedded Linux and connected to a modified Linux host. We empirically evaluate the performance of the Kells device. These experiments indicate that the overheads associated with host validation are minimal, showing a worst case throughput overhead of 1.22% for read operations and 2.78% for writes.

We begin the description of Kells by providing a broad overview of its goals, security model, and operation.

2. OVERVIEW

Figure 1 illustrates the operation of Kells. Once a device is inserted, the host may request a public or trusted partition. If a trusted partition is requested, the host and Kells device perform an attestation-based exchange that validates host integrity. If this fails, the host will be permitted to mount the public partition, if any exists. If the validation process is successful, the host is allowed access to the trusted partition. The process is executed periodically to ensure the system remains in a valid state. The frequency of the re-validation process is determined by the Kells policy.

2.1 Operational Modes

There are two modes of operation for Kells, depending on how much control over device administration should be available to the user and how much interaction he should have with the device. We review these below:

Transparent Mode.

In this mode of operation, the device requires no input from the user. The host verification process executes immediately after the device is inserted into the USB interface. If the process succeeds, the device may be used in a trusted manner as described above, i.e., the device will mount with the trusted partition available to the user. If the attestation process is unsuccessful, then depending on the reason for the failure (e.g., because the host does not contain a TPM or contains one that is unknown to the device), the public partition on the device can be made available. Alternately, the device can be rendered unmountable altogether. A visual indicator on the device such as an LED can allow the user to know whether the host is trusted or not: a green light may indicate a good state while a flashing red light indicates an unknown or untrusted host.

User-Defined Mode.

The second mode of operation provides the user with a more active role in making storage available. When the Kells device is inserted into the system, prior to the attestation taking place, a partition containing user-executable programs is made available. One is a program prompting the user to choose whether to run the device in trusted or public mode. If the user chooses to operate in trusted mode, then the attestation protocol is performed, while if public mode is chosen, no attestations occur. In this manner, the user can make the decision to access either partition, with further policy that may be applied on trusted hosts opening untrusted partitions, to prevent potential malware infection. These hosts may quarantine the public partition, requiring a partition scan prior to allowing access. Such a scan can also be performed by the device. Such a scenario could be useful if there is a need or desire to access specific media (e.g., photographs, songs) from the public partition of the disk while using a trusted host, without having to mark the information as trusted. Trusted partitions on a Kells device are unlikely to be infected to begin with, on account of any host using this partition having to attest its integrity state. This is essential, since a user would not be hesitant to load or execute content from a partition that is considered trusted.

Note that the policies described above are but two examples of the methods of operation available with this infrastructure. For simplicity, we have described the coarse-grained granularity of trusted and public partitions. Within the trusted parition, however, further fine-grained policy can be enforced depending on the identified host; for example, blocks within the partition may be labeled

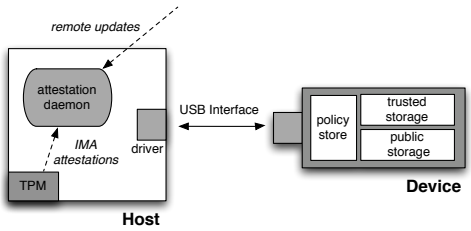


Figure 2: Overview of the Kells architecture.

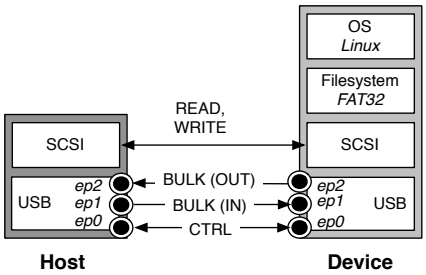


Figure 3: Overview of USB operation with an embedded Linux mass storage device, or *gadget*.

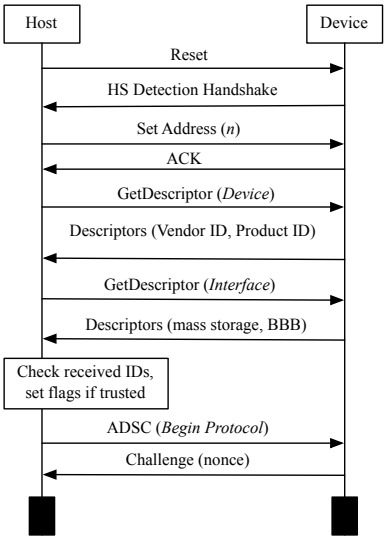


Figure 4: Sample USB setup between a host and the Kells device.

depending on the host writing to them, with a data structure keeping track of the labels and access controls to data (e.g., encrypting labeled data and only decrypting based on the host having access to this label, as specified by device policy).

2.2 Threat Model

We assume the adversary is capable of subverting a host operating system at any time. While we do not specifically address physical attacks against the Kells device, such as opening the drive enclosure to manipulate the physical storage media or modifying the device tick-counter clock, we note that defenses against these attacks have been implemented by device manufacturers. Notably, portable storage devices from IronKey [1] contain significant physical tamper resistance with epoxy encasing the chips on the device, electromagnetic shielding of the cryptographic processor, and a waterproof enclosure. SanDisk's Cruzer Enterprise [24] contains a secure area for encryption keys that is sealed with epoxy glue. Tamper-resistance has also been considered for solutions such as the IBM Zurich Trusted Information Channel [38]. Such solutions would be an appropriate method of defense for Kells. In addition, we assume that any reset of the hardware is detectable by the device (for example, by detecting voltages changes on the USB bus and receiving cleared PCR values from the TPM).

Kells does not in itself provide protection for the host's internal storage, though an adaptation of our design can be used to provide a similar protection mechanism, as with the Firma storage-rooted secure boot system [2]). Integrity-based solutions exist that protect the host's internal storage (hard disks), including storage-based intrusion detection [21] and rootkit-resistant disks [3]. As is common in these systems, we declare physical attacks against the host's TPM outside the scope of this work. As previously discussed, the TPM is used as an implementation point within our architecture and other solutions for providing host-based integrity measurement may be used. As a result, we do not make any attempt to solve the many limitations of TPM usage in our solution. Additionally, we do not consider the issue of devices attesting their state to the host. The TCG's Opal protocol [37] includes provisions for trusted peripherals, addressing the issue by requiring devices to contain TPMs. Software-based attestation mechanisms such as SWATT [27], which does not require additional trusted hardware, may also be used. Finally, we rely on system administrators to provide accurate measurements of their systems, which must be updated if there are changes (e.g., due to configuration or updates). Without updates, Kells will not be able to provide access to the trusted partitions of these systems.

3. DESIGN AND IMPLEMENTATION

We now turn to our design of the Kells architecture, shown in Figure 2, and describe details of its implementation. There are three major components of the system where modifications are necessary: the interface between the host and the device, the storage device itself, and the host's operating system.

3.1 USB Interface

We begin by describing the basics of USB operation in order to aid in understanding the design of Kells. This is a brief overview; more details may be found in the appendix.

The basic USB mass storage device stack is shown in Figure 3. At the USB layer are *endpoints*, the means by which USB commands are send and received. USB mass storage devices primarily use *bulk* transfers, a best-effort service, to transmit information, but every device also supports the flow of *control* transfers over endpoint 0. Above the USB layer is a SCSI emulation layer supporting a limited subset of SCSI commands, such as reads and writes.

Within operating systems that support USB, such as Linux, the number and functionality of supported devices is large and diverse. To support devices (or *gadgets*) that do not conform to the USB specificaton, Linux contains a repository for these devices and sets flags when they to modify the host behavior, in order to correctly operate with these devices.

USB is a master-slave protocol, meaning that all commands must be initiated by the host. This model is conceptually at odds with a device such as Kells, which independently enforces security policy. Therefore, we reconsider how the device interacts with the host.

Figure 4 gives an abridged overview of the device setup process at the USB layer. As with any USB device, high-speed detection

and bus addressing is performed before information is requested by the host. The host requests the device descriptor, which includes information such as the device's vendor and product ID, as well as a unique serial number. When the host requests the interface descriptor, the Kells device identifies itself as a mass storage device that operates in the typical fashion of only performing bulk transfers. The host will set flags accordingly in order to send the correct commands to the device.

Almost every USB mass storage device performs its operations using only bulk transfers. However, we use control transfers for sending trusted commands to Kells. Control transfers reserve a portion of USB bus bandwidth, ensuring that information is tranferred as quickly as possible. If a Kells device is plugged into a host that does not support attestation operations, the host will access the public partition as a standard mass storage device, oblivious to the trusted protocols and storage. If the host recognizes the device as trusted, it will send an Accept Device-Specific Command (ADSC). The setup phase of the command allows the host to initiate the attestation protocol, while the attestation information is sent through the data stage, and the gadget sets a response code that includes a challenge. Further stages of the attestation protocol continue as control transfers between the host and device, and all other read and write operations are suspended until the protocol completes. The attestation protocol is described in detail in Section 4.1.

3.2 Designing the Storage Device

Kells requires the ability to perform policy decisions independent of the host. As a result, logic must execute on these devices, which require a means of receiving command transfers from the host and to use these for making the correct access decisions.

The basic architecture for the storage device is an extension to the Linux USB gadget stack, along with a user-space daemon that is in charge of policy decisions and accessing other important information. Within the kernel, we added new functionality that allows the device to receive special control transfers from the host. These are exported to user space through the *sysfs* interface, where they are read as strings by the daemon tasked with marshaling this data.

When plugged in, the daemon on the device sets a timer (as USB devices contain a crystal oscillator for driving clock signals), and waits to determine whether the host presents the proper credentials. The device presents itself to the host as a typical mass storage device operating in bulk-only mode, differentiating itself with the vendor ID. We use the vendor ID `b000` which has not been currently allocated by the USB Forum as of June 2010.[2]

If an ADSC command containing authenticating information from the host is not received within this time period, operation on the device defaults to public operation. If the device is configured such that the policy does not allow any partitions to be mounted, the device will not present any further information to the host. If the protocol fails, the failure is logged in the storage device's audit log, which is unexposed to the host. Depending on the defined policy, either the public partition will be exposed or no partitions on the device will be mounted at all. If the protocol is successful and the host attests its state to the device, the daemon presents the trusted partition to be mounted, by performing an `insmod()` command to link the correct backing store with the gadget driver.

Within the Kells device is a policy store, which contains information on every known host, its *measurement database* to compare attestations against, and policy details, such as whether the host is authenticated as an administrative console and whether the host should expose a public partition if the attestation check fails. Optionally, the device can also store information on users credentials supplied directly to the device through methods such as biometrics. Configured policy can allow or disallow the device to be plugged into specific machines.

[2]Because this is a proof of concept design and implementation, we have not registered a vendor ID with the USB Forum yet; however, based on our results, we may consider doing so.

3.3 Modifications to Host

A host must be capable of recognizing that the Kells device is trusted and sending information to it differs from a standard USB mass storage transaction. Our goal was to require minimal changes to the host for operation, but because we are working at the USB layer, some changes are necessary to the USB driver. We define a flag `IS_TRUSTED` in the Linux *unusual_devs.h* device repository, letting the host know that the device accepts control transfers.

Because the host must interact with its trusted hardware and perform some logic, we designed an *attestation daemon* that runs in the host's user space. The attestation daemon both retrieves boottime attestations using the Linux Integrity Measurement Architecture (IMA) [23] and can act as an interface to any runtime monitoring systems on the host (see Section 4.2). It can also provide an interface for receiving third-party updates (see Section 4.3).

4. ATTESTATIONS AND ADMINISTRATION

A key consideration with Kells is managing metadata and credential information in a manner that maintains usability and simplicity of the device. We describe in this section details of how this management occurs.

4.1 Attesting Host Integrity

In order for a host connecting to the Kells device to be trustworthy, it must be installed and maintained in a manner that protects its integrity. A way of ensuring this is through the provisioning of a secure kernel and supporting operating system, from which measurements of system integrity can be made and transferred to the Kells device. The maintainer of the host system is thus required to re-measure the system when it is installed or when measurable components are updated. Solutions for ensuring a trusted base installation include the use of a root of trust installer (ROTI) [33], which establishes a system whose integrity can be traced back to the installation media.

The system performing the installation must contain trusted hardware such as a TPM. Every TPM contains an *endorsement key* (EK), a 2048-bit RSA public/private key pair created when the chip is manufactured. This provides us with a basis for establishing the TPM's unique identity, essential to verifying the installation. The stages of this initial installation are as follows:

1. The installation media is loaded into the installer system, which contains a TPM. This system needs to be trusted, i.e., the hardware and system BIOS cannot be subverted at this time.[3] As described below, the system's *core root of trust for measurement* (CRTM), containing the boot block code for the BIOS, provides a self-measurement attesting this state.

2. A measurement of each stage of the boot process is taken. Files critical to the boot process are hashed, and the list of hashes kept in a file that is *sealed* (i.e., encrypted) by the TPM of the installing system. This process links the installing TPM with the installed code and the filesystem. A Kells device in *measurement mode* can record the measurements from the system, or this can be performed in another

[3]This restriction is not necessary after installation, as malicious changes to the system state will be measured by the CRTM.

manner and transferred to the device at a later time, through placement of the list of hashes in a secure repository.

We first identify the host's TPM. While the EK is unique to the TPM, there are privacy concerns with exposing it. Instead, an *attestation identity key* (AIK) public/private key pair is generated as an alias for the EK, and strictly used for signatures. However, the AIK is stored in volatile memory. Therefore, both the public and private AIKs must be stored. The TPM provides the *storage root key* (SRK) pair for encrypting keys stored outside the TPM. Thus, the SRK encrypts the private AIK before it is sent to the device. Formally, the set of operations occurs as follows. Given a host's TPM H and a device D, the following protocol flow describes the initial pairing of the host to the device and the initial boot:

Pairing

(1) H : generate AIK = (AIK^+, AIK^-)

(2) $H \rightarrow D : AIK^+, \{AIK^-\}_{SRK-}$

Measurement

(3) $D \rightarrow H : \{AIK^-\}_{SRK-}$

(4) $D : n =$ Generate nonce

(5) $D \rightarrow H : Challenge(n)$

(6) $H \rightarrow D : Attestation = Quote + ML$

(7) $D : Validate(Quote, ML)_{AIK+}$

Steps 1 and 2 occur when the host has been initially configured or directly after an upgrade operation, to either the hardware or to files that are measured by the IMA process. Subsequent attestations use this list of measurements, which may also be disseminated back to the administrator and stored with the AIK information so as to allow for remote updates, discussed further in Section 4.3.

The following states are measured in order: (a) the core root of trust for measurement (CRTM), (b) the system BIOS, (c) the bootloader (e.g., GRUB) and its configuration, and (d) the OS. Measurements are made by with the TPM's *extend* operation, which hashes code and/or data, concatenates the result with the previous operation, and stores the result in the TPM's *Platform Configuration Registers* (PCRs). The *quote* operation takes the challenger's nonce n and returns a signature of the form $Sign(PCR, N)_{AIK-}$, when the PCRs and n are signed by the private AIK. The measurement list (ML), which contains a log of all measurements sent to the TPM, is also included.

The above protocol describes a *static* root of trust for measurement, or SRTM. There are some disadvantages to this approach, since the BIOS must be measured and any changes in hardware require a new measurement; additionally, it may be susceptible to the TPM reset attack proposed by Kauer [13]. Another approach is to use a *dynamic* root of trust for measurement (DRTM), which allows for a *late launch*, or initialization from a secure loader after the BIOS has loaded, so that it does not become part of the measurement. SRTM may be vulnerable to code modification if DRTM is supported on the same device [6]. DRTM may also be potentially vulnerable to attack; the Intel TXT extensions supporting DRTM may be susceptible to System Management Mode on the processor being compromised before late launch is executed, such that it becomes part of the trusted boot and is not again measured [39]. For this reason, it is an administrative decision as to which measurement mode the system administrator should use for their system, but we can support either approach with Kells.

Note that we are directly connecting with the host through the physical USB interface. The cuckoo attack described by Parno [20] may be mitigated by turning off network connectivity during the boot-time attestation process, such that no remote TPMs can answer in place of the host. However, if the host can access an oracle that presents TPM-like answers, a means to uniquely identify the host is necessary. We are actively investigating these methods.

4.2 Managing Runtime Integrity Attestations

```
1: (att, t) ← read.RAM.att
2: if |req.time − t| < Δ_t ∧ GoodAtt(att) then
3:     Perform the write req as usual.
4: else
5:     if WriteBuffer.notFull() then
6:         Buffer the request for later write back once a fresh attestation
           is received.
7:     else
8:         Stall until there is space in the write buffer.
9:     end if
10: end if
```

Figure 5: Write(req) algorithm.

```
1: (att, t) ← read.RAM.att
2: if GoodAtt(att) then
3:     for Requests buffered before t do
4:         Perform the write req as usual.
5:     end for
6: end if
```

Figure 6: Commit() algorithm.

```
1: (att, t) ← read.RAM.att
2: if |req.time − t| < Δ_t ∧ GoodAtt(att) then
3:     Perform the read req as usual.
4: else
5:     Stall until a fresh attestation is received.
6: end if
```

Figure 7: Read(req) algorithm.

To perform authentication of the host, the Kells device must compare received attestations with a known set of good values. A portion of non-volatile memory is used for recording this information, which includes a unique identity for the host (e.g., the public AIK) the host's measurement list, and policy-specific information, (e.g., should the host allow administrative access).

We provide a framework for supporting runtime integrity monitoring, but we do not impose constraints on what system is to be used. The runtime monitor can provide information to the storage device as to the state of the system, with responses that represent good and bad system states listed as part of the host policy. Our design considers attestations from a runtime monitor to be delivered in a consistent, periodic manner; one may think of them as representing a *security heartbeat*. The period of the heartbeat is fixed by the device and transmitted to the host as part of the device enumeration process, when other parameters are configured.

Because the device cannot initiate queries to the host, it is incumbent on the host to issue a new attestation before the validity period expires for the existing one. The Kells device can issue a warning to the host a short time period λ before the attestation period Δ_t expires, in case the host neglects to send the new attestation.

Algorithms 5 and 6 describe the write behavior on the device. We have implemented a buffer for writes that we term a *quarantine buffer*, to preserve the integrity of data on the Kells device. Writes are not directly written to the device's storage but are stored in the

buffer until an attestation arrives from the host to demonstrate that the host is in a good state. Once a successful attestation arrives, the buffer is cleared, but if a failed attestation arrives and access to the trusted partition is revoked, any information in the write buffer at that time will be discarded. In a similar manner, Algorithm 7 describes the semantics of the read operation. Reads occur as normal unless an attestation has not been received within time Δ_t. If this occurs, then further read requests will be prevented until a new successful attestation has been received.

To prevent replay, the host must first explicitly notify Kells that the attestation process is beginning in order to receive a nonce, which is used to attest to the freshness of the resulting runtime attestation (i.e., as a MAC tied to the received message).

4.3 Remote Administration

An additional program running on the host (and measured by the Kells device) allows for the device to remotely update its list of measured hosts. This program starts an SSL session between the running host and a remote server in order to receive new policy information, such as updated measurements and potential host revocations. The content is encrypted by the device's public key, the keypair of which is generated when the device is initialized by the administrator, and signed by the remote server's private key.

Recent solutions have shown that in addition to securing the transport, the integrity state of the remote server delivering the content can be attested [19]. It is thus possible for the device to request the attestation proof from the remote administrator prior to applying the received policy updates.

In order for the device to receive these updates, the device exposes a special administrative partition if an update is available, signaled to do so by the attestation daemon. The user can then move the downloaded update file into the partition, and the device will read and parse the file, appending or replacing records within the policy store as appropriate. Such operations include the addition of new hosts or revocation of existing ones, and updates of metadata such as measurement lists that have changed on account of host upgrades. This partition contains only one other file: the audit failure log is encrypted with the remote server's public key and signed by the device, and the user can then use the updater program to send this file to the remote server. The server processes these results, which can be used to determine whether deployed hosts have been compromised.

5. REASONING ABOUT ATTESTATIONS

We now prove that the Kells design achieves its goal of protecting data from untrusted hosts. This is done using the logic of secure systems (LS^2) as described by Datta et al. in [6]. Using LS^2, we describe two properties, (**SEC**) and (**INT**), and prove that they are maintained by Kells. These two properties assert that the confidentiality and integrity of data on the Kells device are protected in the face of an untrusted host. To prove that Kells enforces the two properties, we first encode the Kells read and write operations from section 4.2 into the special programming language used by LS^2. These encodings are then mapped into LS^2 and shown to maintain both properties. Both properties are stated informally as follows.

1. (**SEC**) Any read request completed by Kells was made while the host was in a known good state. This means that an attestation was received within a time window of Δ_t from the request or after the request without a host reboot.
2. (**INT**) Any write request completed by Kells was made while the host was in a known good state with the same respect to Δ_t as read.

5.1 Logic of Secure Systems

The logic of secure systems (LS^2) provides a means for reasoning about the security properties of programs. This reasoning allows the current state of a system to be used to assert properties regarding how it got to that state. In the original work, this was used to show that given an integrity measurement from a remote host, the history of programs loaded and executed can be verified. In the case of Kells, we use such a measurement to make assertions about the reads and writes between the host system and Kells storage device, namely, that (**SEC**) and (**INT**) hold for all reads and writes. LS^2 consists of two parts: a programming language used to model real systems, and the logic used to prove properties about the behavior of programs written in the language. This section begins with a description of the language used by LS^2, followed by a description of the logic and proof system.

LS^2 uses a simple programming language, hereafter referred to as "the language," to encode real programs. Any property provable using LS^2 holds for all execution traces of all programs written in the language. Our aim is to encode Kells operation in the language and formally state and prove its security properties using LS^2. The main limitation of the language (and what makes it feasible to use for the verification of security properties) is the lack of support for control logic such as if-then-else statements and loops. Expressions in the language resolve to one of a number of data types including numbers, variables, and cryptographic keys and signatures. For Kells operation, we use numeric values as timestamps (t) and data (n), and pairs of these to represent data structures for attestations and block requests. The expressions used for encoding values in Kells is shown in Table 1.

The language encapsulates operations into *actions*, single instructions for modeling system-call level behavior. Program traces are sequences of actions. There are actions for communication between threads using both shared memory and message passing. In the case of shared memory, `read` l and `write` l, e signify the reading and writing of an expression e to a memory location l. As Kells adds security checks into these two operations, we introduce language extensions `sread` req, att and `swrite` req, att, which are covered in the following section. Finally, the actions `send` req and `receive` are used to model communication with the host ($\mathcal{H}$) by the Kells device ($\mathcal{D}$).

Moving from the language to the logic proper, LS^2 uses a set of logical predicates as a basis for reasoning about programs in the language. There are two kinds of predicates in LS^2, *action predicates* and *general predicates*. Action predicates are true if the specified action is found in a program trace. Furthermore, they may be defined at a specific time in a program's execution, e.g. $\mathsf{Send}(\mathcal{D}, req)$ @ t holds if the thread $\mathcal{D}$ send the results of the request req to the host at time t. See the predicates in Table 1. General predicates are defined for different system states either at an instant of time or over a period. One example of such a predicate is $\mathsf{GoodState}(\mathcal{H}, (t, \mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig))$, which we defined to show that the host system is in a good state with respect to a particular block request. The exact definition of GoodState is given in the following section.

5.2 Verification of Kells Security Properties

We verify that Kells operations maintain the (**SEC**) and (**INT**) properties in several steps. First, we rewrite the algorithms described in section 4.2 using the above described language. This includes a description about assumptions concerning the characteristics of the underlying hardware and an extension of the language to support the write queueing mechanism, along with the operational semantics of these expressions as shown in Figure 8. We then formally

Table 1: The subset of LS^2 and extensions used to evaluate the Kells security properties.

Expressions

Expression	*Use in Validation*
$att = (\mathtt{t_{att}}, sig)$	An attestation consisting of wall clock arrival time $\mathtt{t_{att}}$, and a signature, sig.
$req = (t, \mathtt{t_{req}}, (l, n))$	A block request consisting of a stored program counter t, a wall clock time $\mathtt{t_{req}}$, a disk location l and a value n.

Language Features (* indicates an extension)

Feature	*Use in Validation*
`send` req	Send the result of request req from Kells to the host.
`receive`	Receive a value from the host.
`proj`$_1$ e	Project the first expression in the pair resulting from e. `proj`$_2$ e projects the second expression.
*`enqueue` req	Enqueue the request req in the Kells request queue.
*`peek`	Peek at the item at the head of the Kells device's write request queue. If the queue is empty, halt the current thread immediately.
*`dequeue`	Dequeue a block request from the Kells request buffer.
*`sread` req, att	Perform a secure (attested) read.
*`swrite` req, att	Perform a secure (attested) write.

Predicates (* indicates an extension)

Predicate	*Use in Validation*
$\mathsf{Send}(\mathcal{D}, req)$ @ t	The Kells disk controller ($\mathcal{D}$) sent the result of request req to the host at time t.
$\mathsf{Recv}(\mathcal{D}, req)$ @ t	The Kells disk controller ($\mathcal{D}$) received the request req from the host at time t.
$\mathsf{Reset}(\mathcal{H})$ @ t	Some thread on the host machine ($\mathcal{H}$) restarted the system at time t.
$^*\mathsf{Peek}(\mathcal{D})$ @ t	The Kells disk controller ($\mathcal{D}$) peeked at the tail of the request queue at time t.
$^*\mathsf{SRead}(req, att)$	`sread` was executed in the program trace.
$^*\mathsf{SWrite}(req, att)$	`swrite` was executed in the program trace.
$^*\mathsf{Fresh}(t, \mathtt{t_{req}}, \mathtt{t_{att}})$	The attestation received at time $\mathtt{t_{att}}$ was received recently enough to be considered fresh w.r.t. a request that arrived at $\mathtt{t_{req}}$.
$^*\mathsf{GoodState}(\mathcal{H}, req, att)$	The host ($\mathcal{H}$) attested a good state w.r.t. the request req. Meaning that the host was in a good state when the request was received.

Configuration (* indicates an extension)

Configuration	*Use in Validation*
σ	The store map of $[location \mapsto expression]$. This is used in the semantics of `read` and `write` as well as the write request queue.
$^*(h, t)$	The Kells requests queue, implemented as a pair of pointers to the memory store σ.
$^*\rho$	The program counter. This counter is initialized to t_0 at reboot time and increments once for each executed action in the trace.

$$\text{(enqueue)} \quad \rho, (h,t), \sigma[t \mapsto \_], [x{:=}\ \mathtt{enqueue}\ e; P]_I \longrightarrow \rho+1, (h, t+1), \sigma[t \mapsto (e, \rho)], [P(0/x)]_I$$

$$\text{(dequeue)} \quad \rho, (h,t), \sigma[h \mapsto e], [x{:=}\ \mathtt{dequeue}; P]_I \longrightarrow \rho+1, (h+1, t), \sigma[h \mapsto e], [P(0/x)]_I$$

$$\text{(peek)} \quad \rho, (h,t), \sigma[t \mapsto e], [x{:=}\ \mathtt{peek}; P]_I \longrightarrow \rho+1, (h,t), \sigma, [P(e/x)]_I$$

$$\text{(sread)} \quad \rho, \sigma[l \mapsto e], [x{:=}\ \mathtt{sread}\ (t, \mathtt{t_{req}}, (l,n)), (\mathtt{t_{att}}, sig)]_I \longrightarrow \rho+1, \sigma[l \mapsto e], [P(e/x)]_I \quad \text{if } \mathsf{GoodState}(\mathcal{H}, (t, \mathtt{t_{req}}, (l,n)), (\mathtt{t_{att}}, sig))$$

$$\text{(swrite)} \quad \rho, \sigma[l \mapsto \_], [x{:=}\ \mathtt{swrite}\ (t, \mathtt{t_{req}}, (l,n)), (\mathtt{t_{att}}, sig)]_I \longrightarrow \rho+1, \sigma[l \mapsto e], [P(0/x)]_I \quad \text{if } \mathsf{GoodState}(\mathcal{H}, (t, \mathtt{t_{req}}, (l,n)), (\mathtt{t_{att}}, sig))$$

$$\text{(sreadD)} \quad \rho, \sigma[l \mapsto e], [x{:=}\ \mathtt{sread}\ (t, \mathtt{t_{req}}, (l,n)), (\mathtt{t_{att}}, sig)]_I \longrightarrow \rho+1, \sigma[l \mapsto e], [P(0/x)]_I \quad otherwise$$

$$\text{(swriteD)} \quad \rho, \sigma[l \mapsto \_], [x{:=}\ \mathtt{swrite}\ (t, \mathtt{t_{req}}, (l,n)), (\mathtt{t_{att}}, sig)]_I \longrightarrow \rho+1, \sigma[l \mapsto \_], [P(0/x)]_I \quad otherwise$$

Figure 8: The operational semantics of the language extensions used to encode Kells operations. The program counter ρ applies to all actions in the language.

state the two properties and show that they hold for the encoded versions of Kells operations.

5.2.1 Encoding Kells Operation

The encoding of the read operation is shown in Figure 9 and the write operation in Figure 10. The primary challenge in encoding Kells operations using the language was the lack of support for conditional statements and loops. Note that their addition would also require an extension of the logic to handle these structures. To alleviate the need for loops, we assume that the Kells device has a hardware timer that can repeatedly call the program that performs commits from the write request queue (`KCommit` in Figure 10).

We extend the language with three instructions for working with the Kells write request queue: `enqueue`, `dequeue` and `peek`. The first two operations are straightforward and are assumed to be synchronized with any other executing threads. The `peek` operation prevents a dequeued request from being lost by `KCommit` if a

```
KRead: 1. att = read D.RAM.att-loc
       2. (t, req) = receive
       3. n' = sread req,att
       4. send n'
```

Figure 9: The encoding of the Kells read operation.

```
KWrite:  1. (t, req-pair) = receive
         2. enqueue (t, req-pair)

KCommit: 1. att = read D.RAM.att-loc
         2. (t, req) = peek
         3. swrite req,att
         4. dequeue
```

Figure 10: The encoding of the Kells write operation.

fresh attestation has not arrived after the request has been dequeued. If the queue is empty, `peek` halts the current thread.

To capture Kells mediation, we add the checks for attestation freshness and verification into the semantics of the `read` and `write` actions by introducing the `sread` and `swrite` actions. The semantics of these two actions are shown in Figure 8. Both of these operations take a block I/O request and an attestation as arguments. A block request $(t, \mathtt{t_{req}}, (l, n))$ from the host consists of the program counter at arrival time t, an absolute arrival time $\mathtt{t_{req}}$ and a sector offset and data pair.

The encoded version of the Kells read program (`KRead`) is shown in Figure 9. We assume the existence of a running thread that is responsible for requesting new attestations from the host at a rate of Δ_t and placing the most recent attestation at $\mathcal{D}$`.RAM.att-loc`. Lines 1. and 2. receive the attestation and request from the host respectively. Line 3. invokes the secure read operation which runs to completion returning either the desired disk blocks (sread) or an error (sreadD). Line 4. sends the resulting value to the host.

The encoded version of the Kells write program (`KWrite`) is shown in Figure 10. `KWrite` simply receives the request from the host in line 1. and places it in the request queue at line 2. t contains the value of ρ at the time the request was received. The majority of the write operation is encoded in `KCommit`, which retrieves an enqueued request, arrival time and the most recent attestation, and performs an `swrite`. Recall that `KCommit` runs once in a thread invoked by a timer since a timed loop is not possible in LS^2.

5.2.2 Proof of Security Properties

The (**SEC**) and (**INT**) properties may be stated formally as shown in Figures 11 and 12. Both properties ultimately make an assertion about the state of a host at the time it is performing I/O using the Kells device. GoodState, defined in Figure 13, requires that an attestation (1) is fresh with respect to a given block I/O request and (2) represents a trusted state of the host system. In the following two definitions, Δ_t represents the length of time during which an attestation is considered fresh past its reception. Thus, GoodState can be seen as verifying the state of the host w.r.t. a given I/O request, independent of the state at any previous requests.

We use the predicate $\mathsf{Fresh}(t, \mathtt{t_{req}}, \mathtt{t_{att}})$ to state that an attestation is fresh w.r.t. a given request. The attestation is received at wall clock time $\mathtt{t_{att}}$ and the request at time $\mathtt{t_{req}}$. Attestations are received at the t^{th} clock tick, as obtained using the program counter ρ. As described above, Kells will check if a previous attestation is still within the freshness parameter Δ_t before stalling the read or queueing the write. This is the first case in the definition of Fresh in Figure 14. If a request is stalled, the next attestation received is verified before satisfying the request. In this case, a Reset must not occur between the receipt of the request and the check of the next attestation.

$$
\begin{aligned}
(\mathbf{SEC}) \vdash\ & \forall\ (\mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig), t \text{ s.t.} \\
& (\mathtt{t_{req}}, (l, n)) = \mathsf{Recv}(\mathcal{D}) \mathbin{@} t \\
& \wedge\ (\mathtt{t_{att}}, sig) = \mathsf{Recv}(\mathcal{D}) \\
& \wedge\ e = \mathsf{SRead}(\mathcal{D}, (t, \mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig)) \\
& \supset \mathsf{GoodState}(\mathcal{H}, (t, \mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig))
\end{aligned}
$$

Figure 11: Definition of Kells secrecy property.

$$
\begin{aligned}
(\mathbf{INT}) \vdash\ & \forall\ (t, \mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig) \text{ s.t.} \\
& (t, \mathtt{t_{req}}, (l, n)) = \mathsf{Peek}(\mathcal{D}) \\
& \wedge\ (\mathtt{t_{att}}, sig) = \mathsf{Recv}(\mathcal{D}) \\
& \wedge\ \mathsf{SWrite}(\mathcal{D}, (t, \mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig)) \\
& \supset \mathsf{GoodState}(\mathcal{H}, (t, \mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig))
\end{aligned}
$$

Figure 12: Definition of Kells integrity property.

Theorem 1. `KRead` maintains the (**SEC**) security property.
Proof.
Assume that the following holds for an arbitrary program trace.

$$
\begin{aligned}
& \exists\ (\mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig), t, e \text{ s.t.} \\
& \quad (\mathtt{t_{req}}, (l, n)) = \mathsf{Recv}(\mathcal{D}) \mathbin{@} t \\
& \wedge\ (\mathtt{t_{att}}, sig) = \mathsf{Recv}(\mathcal{D}) \\
& \wedge\ e = \mathsf{SRead}((t, \mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig))
\end{aligned}
$$

We know that t is the value of ρ at the time the request was received because we assumed Recv occurred in the trace at time t. By definition of SRead, we have $\mathsf{Fresh}(t, \mathtt{t_{req}}, \mathtt{t_{att}})$, $\mathsf{Verify}((\mathtt{t_{att}}, sig), AIK(\mathcal{H}))$, and $\mathsf{Match}(v, criteria)$ all hold. Thus, GoodState holds, and (**SEC**) is provable using LS^2 with extensions. Because `KRead` is implemented in the language with extensions, (**SEC**) holds over `KRead` by the soundness property of LS^2.

Theorem 2. `KCommit` maintains the (**INT**) security property.
Proof.
Assume that the following holds for an arbitrary program trace.

$$
\begin{aligned}
& \exists\ (t, \mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig) \text{ s.t} \\
& \quad (t, \mathtt{t_{req}}, (l, n)) = \mathsf{Peek}(\mathcal{D}) \\
& \wedge\ (\mathtt{t_{att}}, sig) = \mathsf{Recv}(\mathcal{D}) \\
& \wedge\ \mathsf{SWrite}((t, \mathtt{t_{req}}, (l, n)), (\mathtt{t_{att}}, sig))
\end{aligned}
$$

We know that t is the value of ρ at the time the request was received, by (enqueue). By definition of SWrite, we have that $\mathsf{Fresh}(t, \mathtt{t_{req}}, \mathtt{t_{att}})$, $\mathsf{Verify}((\mathtt{t_{att}}, sig), AIK(\mathcal{H}))$, and $\mathsf{Match}(v, criteria)$ all hold. Thus, GoodState, holds, giving that (**INT**) is provable using LS^2 with extensions. Because `KCommit` is implemented in the language with extensions, (**INT**) holds over `KCommit` by the soundness property of LS^2.

6. EVALUATION

We performed a series of experiments aimed at characterizing the performance of Kells in realistic environments. All experiments were performed on a Dell Latitude E6400 laptop running Ubuntu 8.04 with the Linux 2.6.28.15 kernel. The laptop TPM performs a single quote in 880 msec. The Kells device was implemented using a DevKit 8000 development board that is largely a clone of the popular BeagleBoard.[4] The board contains a Texas Instruments

[4]Due to extreme supply shortages, we were unable to procure a BeagleBoard or our preferred platform, a small form-factor Gum-

Configuration (Δ_t)	Read			Write		
	Run (secs)	Throughput (MB/sec)	Overhead	Run (secs)	Throughput (MB/sec)	Overhead
No verification	36.1376	14.196	N/A	35.4375	5.6437	N/A
1 second verification	36.5768	14.025	1.22%	36.4218	5.4912	2.78%
2 second verification	36.6149	14.011	1.32%	35.9895	5.5572	1.56%
5 second verification	36.3143	14.127	0.49%	35.7969	5.5871	1.01%
10 second verification	36.2113	14.167	0.20%	35.7353	5.5967	0.84%

Table 2: Kells performance characteristics – average throughput over bulk read and write operations

$$\begin{aligned}&\mathsf{GoodState}(\mathcal{H}, (t, \mathtt{t}_{\mathtt{req}}, (l, n)), (\mathtt{t}_{\mathtt{att}}, sig)) = \\ &\quad \mathsf{Fresh}(t, \mathtt{t}_{\mathtt{req}}, \mathtt{t}_{\mathtt{att}}) \\ &\wedge\ v = \mathsf{Verify}((\mathtt{t}_{\mathtt{att}}, sig), AIK(\mathcal{H})) \\ &\wedge\ \mathsf{Match}(v, criteria)\end{aligned}$$

Figure 13: Definition of Goodstate property.

$$\begin{aligned}&\mathsf{Fresh}\ (t, \mathtt{t}_{\mathtt{req}}, \mathtt{t}_{\mathtt{att}}) = \\ &\quad (\mathtt{t}_{\mathtt{att}} < \mathtt{t}_{\mathtt{req}} \wedge\ \mathtt{t}_{\mathtt{req}} - \mathtt{t}_{\mathtt{att}} < \Delta_t) \\ &\vee\ (\mathtt{t}_{\mathtt{req}} < \mathtt{t}_{\mathtt{att}} \wedge\ \neg\mathsf{Reset}(\mathcal{H}) \text{ on } [t, \rho])\end{aligned}$$

Figure 14: Definition of Fresh property.

OMAP3530 processor, which contains a 600 MHz ARM Cortex-A8 core, along with 128 MB of RAM and 128 MB of NAND flash memory. An SD card interface provides storage and, most importantly for us, the board supports a USB 2.0 On-the-Go interface attached to a controller allowing device-mode operation. The device runs an embedded Linux Angstrom distribution with a modified 2.6.28 kernel. Note that an optimized board could be capable of receiving its power from the bus alone. The TI OMAP-3 processor's maximum power draw is approximately 750 mW, while a USB 2.0 interface is capable of supplying up to 500 mA at 5 V, or 2.5 W. The recently introduced USB 3.0 protocol will be even more capable, as it is able to supply up to 900 mA of current at 5 V.

Depicted in Table 2, our first set of experiments sought to determine the overhead of read operations. Each test read a single 517 MB file, the size of a large video, from the Kells device. We varied the security parameter Δ_t (the periodicity of the host integrity revalidation) over subsequent experiments, and created a baseline by performing the read test with a unmodified DevKit 8000 USB device and Linux kernel. All statistics are calculated from an average of 5 runs of each test.

As illustrated in the table, the read operation performance is largely unaffected by the validation process. This is because the host preemptively creates validation quotes and delivers them to the device at or about the time a new one is needed (just prior to a previous attestation becoming stale). Thus, the validation process is mostly hidden by normal read operations. Performance, however, does degrade slightly as the validation process occurs more frequently. At about the smallest security parameter supportable by the TPM hardware ($\Delta_t = 1\ second$), throughput is reduced by only 1.2%, and as little as 0.2% at 10 seconds. This overhead is due largely to overheads associated with receiving and validating the integrity proofs (which can be as large as 100KB).

Also depicted in Table 2, the second set of tests sought to characterize write operations. We performed the same tests as in the read experiments, with the exception that we wrote a 200MB file. Write operations are substantially slower on flash devices because of the underlying memory materials and structure. Here again, the write operations were largely unaffected by the presence of host validation, leading to a little less than 3% overhead at $\Delta_t = 1\ second$ and just under 1% at 10 seconds.

Note that the throughputs observed in these experiments are substantially lower than USB 2.0 devices commonly provide. USB 2.0 advertises maximal throughput of 480Mbps, with recent flash drives advertising as much as 30MB/sec. All tests are performed on our proof of concept implementation on the experimental apparatus described above, and are primarily meant to show that delays are acceptable. Where needed, a production version of the device and a further optimized driver may greatly reduce the observed overheads. Given the limited throughput reduction observed in the test environment, we reasonably expect that the overheads would be negligible in production systems.

7. RELATED WORK

The need to access storage from portable devices and the security problems that consequently arise is a topic that has been well noted. SoulPad [4] demonstrated that the increasing capacity of portable storage devices allows them to carry full computing stacks that required only a platform to execute on. DeviceSniffer [35] further considered a portable USB device that allowed a kiosk to boot, where the software on the drive provides a root of trust for the system. As additional programs are loaded on the host, they are dynamically verified by the device through comparison with an on-board measurement list. This architecture did not make use of trusted hardware and is thus susceptible to attacks at the BIOS and hardware levels. The iTurtle [17] was a proposal to use a portable device to attest the state of a system through a USB interface. The proposal made the case that load-time attestations of the platform was the best approach for verification. This work was exploratory and postulated questions rather than providing concrete solutions.

Garriss et al. further explored these concepts to use a mobile device to ensure the security of the underlying platform, using it as a kiosk on which to run virtual machines [10] and providing a framework for trusted boot. This work makes different assumptions about how portable devices provide a computing environment; in the proposed model, a mobile phone is used as authenticator, relying on a barcode attached to the platform transmitted wirelessly to the device. Because the verifier is not a storage device, the virtual machine to be run is encrypted in the cloud.

Others have considered trusted intermediaries that establish a root of trust external to the system, starting with Honeywell's Project Guardian and the Scomp system, which provided a secure front-end processor for Multics [8]. SIDEARM was a hardware processor that ran on the LOCK kernel, establishing a separate security enforcement point from the rest of the system [31]. The first attempt to directly interpose a security processor within a system was the Security Pipeline Interface [12], while other initiatives such as the Dyad processor [40] and the IBM 4758 coprocessor [7] provided a secure boot. Secure boot was also considered by Arbaugh et al., whose AEGIS system allows for system startup in the face of in-

stix Overo device. Future work will consider how these devices may change our performance characteristics.

tegrity failure. Numerous proposals have considered how to attest system state. SWATT [27] attests an embedded device by verifying its memory through pseudorandom traversal and checksum computation. This requiers verifier to fully know the memory contents. Recent work has shown that SWATT may be susceptible to return-oriented rootkits [5] but this work itself is subject to assumptions about SWATT that may not be valid. Similarly, Pioneer [26] enables software-based attestation through verifiable code execution by a verification function, reliant on knowledge of the verified platform's exact hardware configuration. A study of Pioneer showed that because it is based on noticing increases in computation time in the event of code modification, a very long execution time is required in order to find malicious computation as CPU speeds increase [9]. Software genuinity [14] proposed relying on the self-checksumming of code to determine whether it was running on a physical platform or inside a simulator; however, Shankar et al. showed problems with the approach [29].

Augmenting storage systems to provide security has been a topic of sustained interest over the past decade. Initially, this involved network-attached secure disks (NASD) [11], an infrastructure where metadata servers issue capabilities to disks augmented with processors. These capabilities are the basis for access control, requiring trust in servers external to the disk. Further research in this vein included self-securing storage [34], which, along with the NASD work, considered object-based storage rather than the block-based approach that we use. Pennington et al. [21] considered the disk-based intrusion detection, requiring semantically-aware disks [30] for deployment at the disk level.

8. CONCLUSION

In this paper, we presented Kells, a portable storage device that validates host integrity prior to allowing read or write access to its contents. Access to trusted partitions is predicated on the host providing ongoing attestations as to its good integrity state. Our prototype demonstrates that overhead of operation is minimal, with a reduction in throughput of 1.2% for reads and 2.8% for writes given a one-second periodic runtime attestation. Future work will include a detailed treatment of how policy may be enforced in an automated way between trusted and untrusted storage partitions, and further interactions with the OS in order to support and preserve properties such as data provenance and control of information flow.

9. REFERENCES

[1] IronKey. `http://www.ironkey.com`, 2009.

[2] K. Butler, S. McLaughlin, T. Moyer, J. Schiffman, P. McDaniel, and T. Jaeger. Firma: Disk-Based Foundations for Trusted Operating Systems. Technical Report NAS-TR-0114-2009, Penn State Univ., Apr. 2009.

[3] K. R. B. Butler, S. McLaughlin, and P. D. McDaniel. Rootkit-Resistant Disks. In *ACM CCS*, Oct. 2008.

[4] R. Cáceres, C. Carter, C. Narayanaswami, and M. Raghunath. Reincarnating PCs with portable SoulPads. In *ACM MobiSys*, 2005.

[5] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. In *ACM CCS*, Nov. 2008.

[6] A. Datta, J. Franklin, D. Garg, and D. Kaynar. A Logic of Secure Systems and its Application to Trusted Computing. In *IEEE Symp. Sec. & Priv.*, May 2009.

[7] J. G. Dyer, M. Lindermann, R. Perez, et al. Building the IBM 4758 Secure Coprocessor. *IEEE Computer*, 39(10):57–66, Oct. 2001.

[8] L. J. Fraim. Scomp: A solution to the multilevel security problem. *IEEE Computer*, 16(7):26–34, July 1983.

[9] R. Gardner, S. Garera, and A. D. Rubin. On the difficulty of validating voting machine software with software. In *USENIX EVT*, Aug. 2007.

[10] S. Garriss, R. Cáceres, S. Berger, R. Sailer, L. van Doorn, and X. Zhang. Trustworthy and personalized computing on public kiosks. In *ACM MobiSys*, June 2008.

[11] G. A. Gibson, D. F. Nagle, K. Amiri, et al. A Cost-Effective, High-Bandwidth Storage Architecture. In *ASPLOS*, 1998.

[12] L. J. Hoffman and R. J. Davis. Security Pipeline Interface (SPI). In *ACSAC*, Dec. 1990.

[13] B. Kauer. OSLO: Improving the Security of Trusted Computing. In *Proc. USENIX Security Symp.*, Aug. 2007.

[14] R. Kennell and L. H. Jamieson. Establishing the Genuinity of Remote Computer Systems. In *Proc. USENIX Security Symp.*, Aug. 2003.

[15] Kingston Technology. DataTraveler 300: World's first 256 GB Flash drive. `http://www.kingston.com/ukroot/flash/dt300.asp`, July 2009.

[16] C. Lomax. Security tightened as secretary blamed for patient data loss. *Telegraph & Argus*, 4 June 2009.

[17] J. M. McCune, A. Perrig, A. Seshadri, and L. van Doorn. Turtles all the way down: Research challenges in user-based attestation. In *USENIX HotSec*, Aug. 2007.

[18] Microsoft. BitLocker and BitLocker to Go. `http://technet.microsoft.com/en-us/windows/dd408739.aspx`, Jan. 2009.

[19] T. Moyer, K. Butler, J. Schiffman, et al. Scalable Web Content Attestation. In *ACSAC*, 2009.

[20] B. Parno. Bootstrapping trust in a "trusted" platform. In *USENIX HotSec*, Aug. 2008.

[21] A. G. Pennington, J. D. Strunk, J. L. Griffin, et al. Storage-based Intrusion Detection: Watching storage activity for suspicious behavior. In *Proc. USENIX Security*, 2003.

[22] P. Porras, H. Saidi, and V. Yegneswaran. An Analysis of Conficker's Logic and Rendezvous Points. Technical report, SRI Computer Science Lab, Mar. 2009.

[23] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proc. USENIX Security*, Aug. 2004.

[24] SanDisk. SanDisk Cruzer Enterprise. `http://www.sandisk.com/business-solutions/enterprise`, 2009.

[25] Seagate. Self-Encrypting Hard Disk Drives in the Data Center. Technology Paper TP583.1-0711US, Nov. 2007.

[26] A. Seshadri, M. Luk, E. Shi, et al. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *ACM SOSP*, 2005.

[27] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: SoftWare-based ATTestation for Embedded Devices. In *IEEE Symp. Sec.& Priv.*, May 2004.

[28] N. Shachtman. Under Worm Assault, Military Bans Disks, USB Drives. *Wired*, Nov. 2008.

[29] U. Shankar, M. Chew, and J. D. Tygar. Side Effects are Not Sufficient to Authenticate Software. In *Proc. USENIX Security*, 2004.

[30] M. Sivathanu, V. Prabhakarn, F. I. Popovici, et al. Semantically-Smart Disk Systems. In *USENIX FAST*, 2003.

[31] R. E. Smith. Cost profile of a highly assured, secure operating system. *ACM Trans. Inf. Syst. Secur.*, 4(1):72–101, 2001.

[32] SRN Microsystems. Trojan.adware.win32.agent.bz. `http://www.srnmicro.com/virusinfo/trj10368.htm`, 2009.

[33] L. St. Clair, J. Schiffman, T. Jaeger, and P. McDaniel. Establishing and Sustaining System Integrity via Root of Trust Installation. In *ACSAC*, 2007.

[34] J. Strunk, G. Goodson, M. Scheinholtz, et al. Self-Securing Storage: Protecting Data in Compromised Systems. In *USENIX OSDI*, 2000.

[35] A. Surie, A. Perrig, M. Satyanarayanan, and D. J. Farber. Rapid trust establishment for pervasive personal computing. *IEEE Pervasive Computing*, 6(4):24–30, Oct.-Dec. 2007.

[36] TCG. *TPM Main: Part 1 - Design Principles*. Specification Version 1.2, Level 2 Revision 103. TCG, July 2007.

[37] TCG. *TCG Storage Security Subsystem Class: Opal*. Specification Version 1.0, Revision 1.0. Trusted Computing Group, Jan. 2009.

[38] T. Weigold, T. Kramp, R. Hermann, et al. The Zurich Trusted Information Channel – An Efficient Defence against Man-in-the-Middle and Malicious Software Attacks. In *Proc. TRUST*, Villach, Austria, Mar. 2008.

[39] R. Wojtczuk and J. Rutkowska. Attacking Intel Trusted Execution Technology. In *Proc. BlackHat Technical Security Conf.*, Feb. 2009.

[40] B. Yee and J. D. Tygar. Secure Coprocessors in Electronic Commerce Applications. In *Proc. USENIX Wrkshp. Electronic Commerce*, 1995.

Keeping Data Secret under Full Compromise using Porter Devices

Christina Pöpper
System Security Group
Computer Science
ETH Zurich
poepperc@inf.ethz.ch

David Basin
Information Security Group
Computer Science
ETH Zurich
basin@inf.ethz.ch

Srdjan Čapkun
System Security Group
Computer Science
ETH Zurich
capkuns@inf.ethz.ch

Cas Cremers
Information Security Group
Computer Science
ETH Zurich
cremers@inf.ethz.ch

ABSTRACT

We address the problem of confidentiality in scenarios where the attacker is not only able to observe the communication between principals, but can also fully compromise the communicating parties (their devices, not only their long term secrets) after the confidential data has been exchanged. We formalize this problem and explore solutions that provide confidentiality after the full compromise of devices and user passwords. We propose two new solutions that use explicit key deletion and forward-secret protocols combined with key storage on porter devices. Our solutions provide the users with control over their privacy. We analyze the proposed solutions using an automatic verification tool. We also implement a prototype using a mobile phone as a porter device to illustrate how the solution can be realized on modern platforms.

Categories and Subject Descriptors

C.2 [**Computer Systems Organization**]: Computer-Communication Networks; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Unauthorized access*

General Terms

Design, Security

Keywords

Security Protocol, System Security, Full Compromise

1. INTRODUCTION

Confidential communication is a basic security requirement for modern communication systems. Solutions to this problem prevent an attacker that observes the communication between two parties from accessing the exchanged data. We address a related, but harder, problem in a scenario where the attacker is not only able to observe the communication between the parties, but can also fully compromise these parties at some time after the confidential data has been exchanged. If a protocol preserves confidentiality under such attacks, we say that it provides *forward secrecy under full compromise*. This is a stronger notion than forward secrecy [18], which guarantees confidentiality when participants' long-term secrets (but not their devices or passwords) are compromised. For example, a subpoena is issued and the communication parties must relinquish their devices and secrets after (e. g., e-mail) communication took place. In this scenario, the parties would like to guarantee that the authorities cannot access the exchanged information, even when given full access to devices, backups, user passwords, and keys, including all session keys stored on the devices.

Assuming public communication channels, any solution to the above problem must ensure that the communication is encrypted to prevent eavesdropping. The challenge in solving this problem is the appropriate management and deletion of the keys used to encrypt the data. Several solutions to this problem have been proposed. First, the Ephemerizer system [28] stores the encryption keys on a physically separate, trusted server accessible by all communicating parties. A drawback of this approach is that trust is placed in one entity, whose compromise would be disastrous for all parties using its services (e. g., companies and individuals). To address this concern, [21] proposes using Distributed Hash Table (DHT) networks for key storage and deletion, thereby removing trust from a central entity. This system, however, only provides probabilistic key deletion without guarantees on the deletion times of stored keys. Furthermore, researchers have shown how to attack this prototype implementation using Sybil attacks on DHTs, which enabled the attackers to reconstruct keys [36]. This attack highlights the problem of delegating key deletion to arbitrarily selected, untrusted nodes.

In this work, we formalize the problem of forward secrecy under full compromise and explore new solutions that provide confidentially after the compromise of devices and user passwords. Our solutions rely on the existence of trusted, reliable *porter devices* that manage encryption keys. We do not require that the principals trust one central server but enable the receivers to select their own key storage devices (based on their trust). We thus enable users to control their own privacy. Although it might seem that – given trusted porter devices – solutions to this problem would be simple, they turn out to be surprisingly complex. This complexity stems from (i) the need to ensure that the protocols do not allow key reconstruc-

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

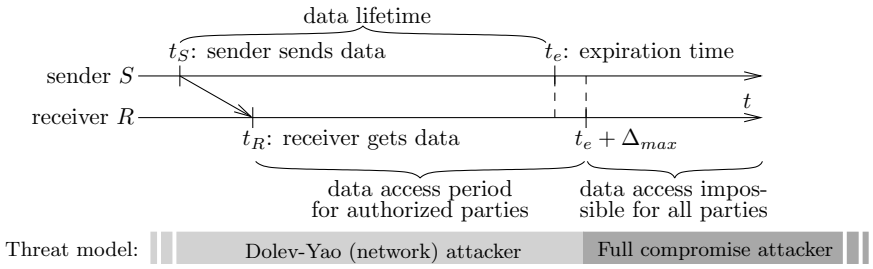


Figure 1: Timeline for time-limited data. Data can be accessed until its expiration time $t_e + \Delta_{\max}$, where t_e relates to the sender's clock and Δ_{max} is the maximal clock difference of the receiver with respect to the sender. After time $t_e + \Delta_{\max}$, data must be inaccessible to all parties, even under full system compromise.

tion under full compromise and (ii) the need to provide guarantees on the key deletion. Abstractly, our solutions use forward-secret subprotocols, session keys with different lifetimes, and timed, explicit key deletion as building blocks to achieve forward secrecy under full compromise. This prevents data access by all parties, including attackers, after a well-defined time. The requirement of guaranteed deletion motivates our use of porter devices: they enable timely key deletion even if the communication devices (e. g., PCs, laptops) cannot be guaranteed to be active.

Our main contributions are as follows. First, we formalize the concept of forward secrecy under full compromise. Second, we present two practical solutions to achieve it. Third, we formally analyze the presented solutions using an automatic verification tool [15]. Finally, we analyze their practical feasibility with a prototype implementation, using a mobile phone as porter device. We thus illustrate how the solution can be realized on modern platforms and how practical considerations can be handled.

The remainder of this paper is organized as follows. In Section 2, we specify the system requirements and our system and attacker models. In Section 3, we motivate our solution. We present our solution and formally analyze its properties in Section 4. In Section 5, we examine possible realizations and describe our prototype implementation. We discuss related work in Section 6 and draw conclusions in Section 7.

2. SYSTEM SPECIFICATION

2.1 Requirement Specification

Our goal is to design a system that provides data access only during a defined time period and afterward prevents access for all parties. We first introduce some key notions, which are illustrated in Figure 1.

Definition 1. The sender specifies data as *time-limited* by assigning a time after which the data must be inaccessible to the sender, the receiver, and any other party. We denote this time by t_e, also called the *expiration time*.

We note that t_e is relative to the sender's local clock.

Definition 2. During the lifetime of time-limited data, *authorized access* is granted only to parties that the sender selects as authorized to access the data.

Our system shall meet the following security requirement:

R1 **Time-dependent access control**: The time-limited data is *inaccessible* outside of the lifetime period specified by the sender.
   (a) During the data lifetime, only authorized access shall be granted.
   (b) After the data lifetime, no data access is possible for any party. This includes the sender, the receiver, and any compromised party.

We also define a functional requirement:

R2 **Data availability**: Given the successful communication between the sender and an authorized receiver (i. e., messages reach the intended recipient), the receiver can access the data during its lifetime.

2.2 System model

We consider the setting where a sender wants to transfer time-limited data to one or more receivers (the authorized recipients). The transfer may use any communication medium and include different applications, e. g., email exchange or server upload and download. A special case is the local storage of time-limited data as a form of self-communication involving only the sender. We make the following four assumptions:

Trusted communication partners. Communication partners, also called *principals*, follow the protocol. In particular, their devices timely and safely delete[1] data and they do not reveal time-limited data or keys in ways not specified by the protocol. Principals may shut down their communication devices and resume communication later, i. e., their devices need not be online at all times.

Authenticated communication. The sender and the receiver can communicate *authentically*. This may be achieved using pre-shared secret keys or authentic, pre-distributed (long-term) public keys. Pre-shared secrets are used to generate and verify message authentication codes (MACs) whereas long-term public keys are used for signature verification.

Trusted storage device. There exists a *reliable device* with an independent clock used for data (key) storage. Typical instances of such devices are built-in Trusted Platform Modules (TPMs), Hardware Security Modules (HSMs), or any external device, such as mobile phones, PDAs, or (e-banking) smartcards with readers (see also Section 5). Throughout this paper, we call this device a *porter* and denote it P. In our solution, the porter must be trusted in three ways: (i) P supports authentic communication, e. g., using authentic public keys or a physically secure channel, (ii) P supports the confidential storage and retrieval of data (in our protocols by the receiver), and (iii) P is regularly active and can provide autonomous, permanent erasure of stored data at specified times (or its inaccessibility after specified times). In general, the simpler the porter

[1]We assume that the principals use secure deletion [24,33] preventing data restoration.

device, the less complex its key deletion operation will be. At the same time, simple porter devices are, in general, more controllable and less error-prone than complex, general-purpose devices. We thus envision TPMs or dedicated smartcards as porters for corporate use and mobile phones or PDAs for (less critical) private use.

Loose time synchronization. The sender S, the receiver R, and the porter P are *loosely time-synchronized*. The local clock differences between the sender and the other principals at the data expiration time do not exceed Δ_{max}: when S's clock hits t_e, R's and P's clocks are between $t_e - \Delta_{max}$ and $t_e + \Delta_{\max}$. The principals' devices are not required to remain synchronized within Δ_{max} throughout the lifetime of the data but just at the expiration time. Time-limited data should be accessible at least until $t_e - \Delta_{max}$ and be inaccessible after $t_e + \Delta_{max}$.

2.3 Attacker model

We consider a two-phased attacker model. Our main aim is to model attackers capable of *full compromise* (introduced below), which models for example court orders or subpoenas. If such an attacker is present during the data access period, all protocols that require the data to be in the (accessible) device's memory are trivially insecure. We therefore design our protocols to provide security guarantees with respect to a two-phase attacker model (Figure 1): (i) *before* and *during* the data access period (defined by the sender), we consider a strong network (Dolev-Yao [19] type) attacker, and (ii) *after* the data access period, we consider an even stronger attacker capable of full compromise. Let $\mathcal{U}$ be the set of users authorized to access the transmitted data before its expiration time t_e.

Attacker model for $t \leq t_e + \Delta_{\max}$: *Active external attacker*. The attacker controls the network and may eavesdrop, intercept, inject, and block messages, but she has no control over the devices of users from the set $\mathcal{U}$. Users not in $\mathcal{U}$ may collude with the attacker and deviate from the protocol description. This attacker model corresponds to the standard Dolev-Yao model and is applicable to communication systems comprising ISPs, web mail providers, proxies, relay nodes, etc.

Attacker model for $t > t_e + \Delta_{\max}$: *Full Compromise*. In addition to controlling the network, the attacker completely controls the users' devices, including porter devices, and can compromise users' passwords and passphrases. The attacker can access and change all data stored on the devices and backups, possibly supported by court orders or subpoenas that oblige users to disclose data. In particular, she may compromise the principals' keys, including long-term and ephemeral secret keys, and she can coerce users to reveal the passwords used to secure decryption keys. We refer to this model as *full compromise*. This model is stronger than the Dolev-Yao model in that it allows the compromise of *all* data on the devices, including the data protected by user-selected passwords.

This two-phase attacker model is very strong. In many practical settings, the first-phase attacker will be weaker than a Dolev-Yao attacker. For example, it may be reasonable to assume that even in case of a subpoena after $t_e + \Delta_{\max}$, only communication logs were recorded in the phase before $t_e + \Delta_{\max}$ (e. g., by web mail or internet service providers), but no active attack was mounted. Indeed, such attacks often make evidence inadmissible. The concept of a phased attacker model also allows us to define other attackers that are stronger than the Dolev-Yao attacker in the first phase. In some scenarios the attacker might use a cryptographic attack to access principals' long-term secrets before getting full access to the devices at $t_e + \Delta_{\max}$. Although this is not part of our core attacker model, our solution even resists some attacks of this nature.

3. SOLUTION SPACE

In this section, we explore the space of possible systems that meet the requirements given in Section 2. We also introduce and categorize related work and motivate our solution.

Data transmitted over an open network cannot, in general, be explicitly deleted since the sender does not have access to (and may not even be aware of) all existing copies. Hence the sender must encrypt data before transmission to protect its confidentiality. Since an attacker (as defined in Section 2.3) may have full access to all devices after the data expiration time, data must also never be *stored* in plaintext on any device where it could possibly still reside after the time $t_e + \Delta_{\max}$. As principals can communicate authentically, they can use public-key cryptography to establish secret (session) keys over open networks and use the resulting keys to secure subsequent communication. The solution space therefore amounts to different ways of creating, managing, and deleting decryption keys.

Intuitive Approaches. We first look at two approaches for key management and deletion that appear intuitive but are inappropriate as solutions.

(1) The sender and receiver delete the established key immediately after the encryption and decryption phases, respectively.

This approach does not fulfill requirement R1.b (Section 2) if the encrypted data sent by the sender S arrives at receiver R after $t_e + \Delta_{\max}$ or if it never arrives at R (e. g., due to message blocking or delay attacks, transmission failures, or R being offline / inactive). In this case, the pre-agreed key K remains stored on R because the receiver never starts the decryption phase. This reveals the time-limited data under full compromise after $t_e + \Delta_{\max}$.

(2) The sender and receiver delete the key at its lifetime expiration t_e, e. g., using a job or task scheduler such as Cron.

This does not guarantee requirement R1.b because these automated tasks are not guaranteed to succeed. For example, users' personal computers usually have periods of inactivity during which they are turned off or they may have to be handed in for repair. In such cases, R may be turned off at the expiration time and system processes cannot erase expired keys from the device memory and disks.

From the above considerations, we conclude that the key K used to encrypt the time-limited data cannot be stored on either S or R. Hence it must be stored externally.

Related Work. We briefly review selected related work to illustrate relevant parts of the solution space. In the Ephemerizer system [28] and its application to file deletion [29], a physically separate, trusted machine, the *Ephemerizer*, generates and stores the keys used to encrypt and decrypt the data. Users interact with the Ephemerizer in order to retrieve the encryption or decryption keys. A potentially large number of users, for example a company's employees, use the same (logical) key generator and storage.

The authors of Vanish [21] propose using a de-centralized key storage based on peer-to-peer networks and DHTs. In their system, the sender picks a random encryption key, splits it using secret sharing, and stores the key shares in a DHT network from where the receiver can retrieve them as long as they exist. Due to the natural churn in such networks, the keys are eventually deleted.

Solution Dimensions. We identify four properties of key storage devices: (i) storage type, (ii) access options, (iii) level of guarantees for key management, and (iv) scalability. In the remainder of this section we explain these properties and show in Table 1 how they apply to the approaches above and to our solution.

(i) Storage type. The storage may be *centralized* (e. g., a remote server [28]), or *distributed* [21]; distributed storage requires key sharing. While deletion on a centralized storage is a well-

	Ephemerizer [28]	Vanish [21]	Our solution (Section 4)
Storage type	centralized/shared	distributed/shared	personal
Key generation	by the storage server	by S	by S and R or by R
Key deletion	deterministic	probabilistic	deterministic
Access to K	both S and R	both S and R	R (or S)
	over an open network	over an open network	over open/trusted networks
Scalability	scales (special-purpose)	limited (secondary with many users)	scales (special-purpose or secondary with few users per storage)

Table 1: Dimensions of the key storage and their instantiation by different solutions. Our solution allows access to the encryption key K by R but can easily be extended to enable access also by S on a separate storage (belonging to S).

defined operation, providing guarantees on the deletion of (sufficiently many) key shares on distributed storage is challenging.

(*ii*) Access. The storage may be *personal* or *shared*. Personal storage allows exclusive access by either S or R. The access to personal storage may be based on *public* or *secure* channels; an example for the latter are independent storage units within a user's device. Shared storage (e.g., a network server) permits multiple parties to store and retrieve data. We do not consider storage that only S and R can access because it is a special case that could be used to directly transfer time-limited data. The communication channels to access shared storage are typically public. Since the key must be stored in plaintext on shared storage[2], it may allow attackers to collect data before the expiration time and use it later to access the data. The attack [36] on Vanish is an example of this.

(*iii*) Guarantees on key management. Any key storage must store and manage keys and delete them in a timely way. We distinguish between *deterministic* and *probabilistic* key deletion. In contrast to probabilistic key deletion, deterministic key deletion provides guarantees on the times when keys will be deleted; it is typically harder to achieve on complex or distributed systems (e. g., network servers) than on simple, monolithic devices.

(*iv*) Scalability. The storage should provide functionality for a large number of users without substantially degraded performance. We distinguish *special-purpose* storage that can be designed to scale well with the number of users (e. g., [28]) and *secondary storage* that fulfills different primary purposes and, additionally, provides key management. In the latter case, the primary functions may degrade with the additional key management functionality of the storage; in this case, the scalability is limited.

4. OUR SOLUTION

4.1 Solution Overview

As motivated in Section 3, the sender encrypts the time-limited data prior to transmission. The encryption key is established on a per-message basis between S and R using an authenticated Diffie-Hellman (DH) key establishment protocol. In our solution, we relocate the encryption keys to an autonomous porter device under the receiver's control (we do not use a central server because it requires the users' trust and creates a single point of failure). The porter device will independently delete keys once the expiration time of the messages encrypted using those keys is reached. Given that the porter possesses the sole copy of this encryption key at the expiration time and the porter will delete keys when they expire, this approach prevents data access by any party after $t_e + \Delta_{\max}$.

[2]If the decryption key K was encrypted, this would bring us back to the original problem: how and where to store the key. Asymmetric encryption with R's long-term public key would not resist a full-compromise attack after $t_e + \Delta_{\max}$.

A porter-based approach requires elaboration to provide authenticity and forward-secrecy for the connections from the sender to the receiver and between the receiver and the porter. This requires carefully managing multiple short-term keys. In security applications, e. g., off-the-record messaging [3, 11], short-term keys are created on demand and deleted immediately after the data encryption and decryption. Deleting the decryption key after the data transmission is, however, not a solution in our scenario: we must ensure data inaccessibility after the expiration time $t_e + \Delta_{\max}$ even if the sender's message is not received before $t_e + \Delta_{\max}$ (see Section 3).

4.2 Forward Secrecy under Full Compromise

We introduce the notion of forward *forward secrecy under full compromise* and explain why we need it. *Forward secrecy* means that the compromise of the principals' long-term private keys does not compromise past session keys [18, 27]. Our system requires forward secrecy not only under the compromise of long-term keys but also under *full compromise* (as defined in Section 2.3) after the expiration time. Given this extended notion of compromise, we similarly extend the definition of forward secrecy.

Definition 3. A protocol is *forward-secret under full compromise* with respect to time-limited data m if the full compromise of the involved principals and their devices after the data expiration time does not compromise the secrecy of m.

Forward secrecy under full compromise is a stronger property than (standard) forward secrecy because it also accounts for the principals' internal states after the expiration time. As a consequence, time-critical data and the respective encryption keys must be erased from the principal's devices such that they are nonexistent at the expiration time. *Key* and *data deletion* must be part of any protocol that provides forward secrecy under full compromise. A second essential component concerns those parts of the protocol that involve session keys, which we call *subprotocols*, e. g., for key establishment. Forward secrecy under full compromise requires that all subprotocols used to establish session keys for data encryption provide *forward secrecy*.

4.3 Protocol

We now present the main idea of our protocols. We focus on the case where the receiver uses the key storage (rather than the sender). Figure 2 provides a protocol sketch that we will later instantiate with concrete solutions. All delete commands are secure deletions. We consider the following four protocol phases:

1. **Key establishment**: The sender S defines the data lifetime t_e and agrees with the receiver R on the mid-term key K (or a key pair where K is the decryption key). R initiates the safe storage of K along with t_e on the porter P and deletes

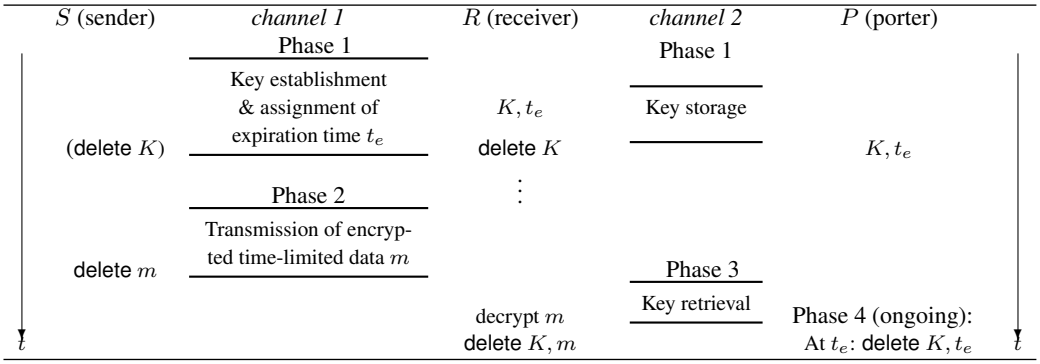


Figure 2: Protocol sketch. The basic building blocks are commands for explicit, secure deletion and forward-secret subprotocols during the communication phases (Phases 1–3).

its own copy of K.[3] If the key establishment involves key contributions from the principals, the ephemeral private keys are deleted right after the key establishment.

2. **Communication/storage**: S transmits the data m, encrypted using key K, and then deletes both the plaintext and K.
3. **Data access**: Upon receiving the encrypted data, R attempts to retrieve K from P in order to decrypt m. After successful data access, R deletes both the plaintext m and K. This phase may occur multiple times.
4. **Key management/deletion**: In parallel with phases 1–3, P permanently deletes keys from its storage once they expire.

Our solution involves three kinds of keys for different time intervals:

1. a *mid-term* encryption key K (or key pair) for encrypting and decrypting time-limited data,
2. *long-term* authentication keys used to authenticate the messages, and
3. *short-term* (ephemeral) session keys to provide secrecy of the communication between the principals and to the porter.

The notion of a *mid-term* key is non-standard but is appropriate for our key K, which must exist during the data's lifetime and is permanently erased thereafter.

Encryption using mid-term keys can be based on symmetric or public-key cryptography. We will provide examples of both in Section 4.4. The examples also differ in the assumptions on the communication channels underlying the protocols. We require two channels: one between the sender and the receiver for data transmission and key-establishment and a second channel between the receiver and the porter for key storage and retrieval. We introduce two common channel types in the following; based on the available channels, different subprotocols will provide forward-secrecy.

Physically secure channel: A physically secure (PS) channel provides confidentiality and authenticity without cryptographic measures. An example of such a channel is a shielded wire that connects the receiver's motherboard to a trusted hardware module. Due to the physical security of the communication, forward secrecy is trivially achieved because no long-term or short-term keys are involved in the communication.

Dolev-Yao channel: A Dolev-Yao (DY) channel is subject to attacks under the Dolev-Yao attacker model, involving eavesdropping, message corruption, insertion, and blocking (erasing). An example of a DY channel is a wireless (e. g., Bluetooth) connection between two devices.

[3]In our protocols, R stores and retrieves the key. In a different protocol, S may also store the key on a porter of its own.

The standard way to achieve forward secrecy on a Dolev-Yao channel is to establish ephemeral encryption keys, typically by using an authenticated DH protocol [12], and to discard them after their use. In this case, the ephemeral DH public keys g^{r_S} and g^{r_R} are exchanged and stored only during the key establishment. They are destroyed thereafter along with the private keys r_S and r_R. The established key $K = g^{r_S r_R} = g^{r_R r_S}$ is the encryption key.

4.4 Protocol Instances

We now present two instances of the protocol sketch of Figure 2, shown in Figures 3 and 4. The two protocols differ in how they achieve forward secrecy on the communication channels between S, R, and P.

We use the following notation: $[M]_K$ and $[M]_K^{-1}$ denote the symmetric encryption and decryption of a message M with key K. $\mathcal{A}_S(M)$ denotes that message M is authenticated by principal S (described below). Communication is expressed as $S \xrightarrow{M} R$, meaning that S transmits message M to receiver R. The tupling of multiple data items in a message is denoted by ",". For DH key establishment, g denotes the public generator of the group used, r_S is the ephemeral private key of principal S, and g^{r_S} is S's ephemeral public key; the use of the modulus (mod n) is implicit.

4.4.1 Protocol 1

Protocol 1 (Figure 3) is designed to be used when S, R, and P communicate over DY channels. In this scenario, P may, e. g., be a mobile phone that belongs to R. Protocol 1 uses symmetric encryption to transmit the time-limited data m. The protocol is initiated by S, who starts a DH key establishment with R. R then establishes another ephemeral DH key L with the porter device P and uses it to send K encrypted to P. Later, after receiving the encrypted time-limited data from S, R establishes a new ephemeral key L' with P and uses L' to retrieve K. For each subsequent retrieval of the encryption key K, a new ephemeral key is established.

The DH key exchanges of Protocol 1 follow the standard two-way ISO-9798-3 protocol [23].[4] We do not require a third message for key confirmation in which the sender returns both ephemeral public keys to the receiver to confirm that it possesses the same key. Under our attacker model, the receiver is not compromised before it sends its DH key contribution (when t_e expires, both parties abort the protocol).

The following components are essential to Protocol 1:

[4]The standard also specifies a random index i into a universal hash function family H in message 2, so that the shared key computed is $K = H_i(g^{r_S r_R})$. We do not use this.

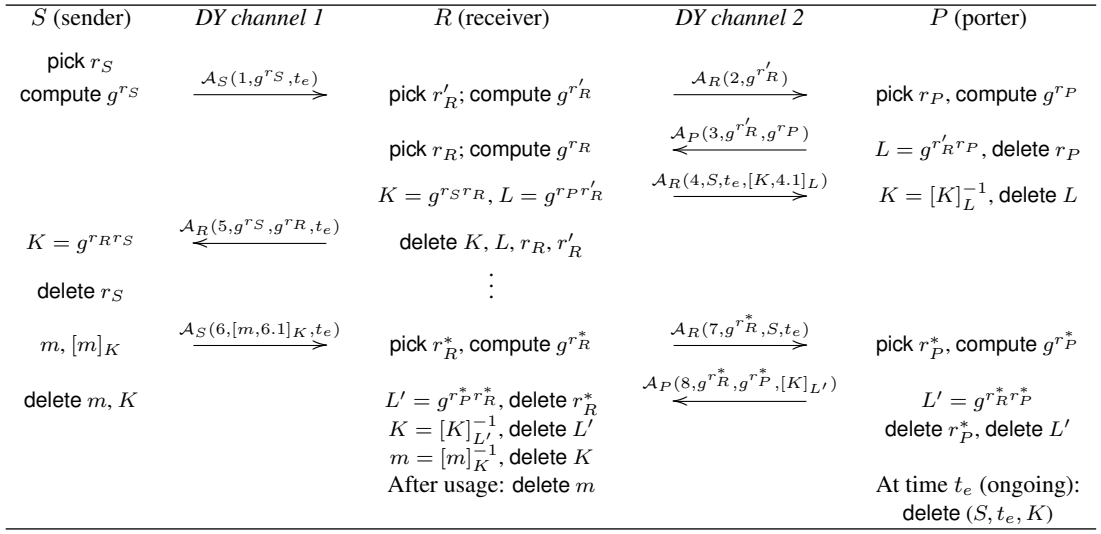


Figure 3: Protocol 1. The protocol can be run over two Dolev-Yao (DY) channels, between S and R and between R and P. The established symmetric mid-term key K is used by S to encrypt the time-limited data m. All messages are authenticated, denoted by the authentication function $\mathcal{A}_X(\cdot)$, which represents the function input concatenated with a digital signature of principal X.

1. All messages are **authenticated** by the transmitter, as indicated by the authentication function $\mathcal{A}_X(\cdot)$ where $X \in \{S, R, P\}$ is the authenticating principal. This can be achieved using message authentication codes (MACs) with pre-shared symmetric keys or by digital signatures using X's long-term secret key. In the latter case, the first message would be $1, g^{r_S}, t_e, Sig_S(1, g^{r_S}, t_e)$, where $Sig_S(M)$ denotes the digital signature of M using S's long-term key.
2. The principals **verify** the authenticity of received messages (by verifying signatures or MACs) and check the validity of t_e. The principals abort the protocol if t_e has expired or if message authenticity cannot be verified.
3. If the protocol aborts due to failed time or authenticity checks, **abortive measures** must be taken. In particular, critical data (such as encryption keys and DH key contributions), which may be present on a device, must be securely deleted.

Encrypted vs. plain storage of K. In Protocol 1, the mid-term encryption (decryption) key is stored in plaintext on P and revealed only to R (encrypted over the DY-channel). While the unencrypted storage of K may seem like a weakness, under our attacker model, full device compromise only occurs after the expiration time when K is already deleted. We still consider it realistic that the porter device (e. g., a mobile phone) may be lost or stolen before the expiration time; in either case, we can assume that the owner is aware of the loss. To preserve data privacy in this case, we propose to store K *encrypted* on P: In Protocol 1, the receiver would send the encrypted key K to the porter (i. e., $[[K]_X]_L$ instead of $[K]_L$) and store the symmetric encryption key X along with the expiration time t_e of K on R. Whenever the owner notices the loss of his porter device, he can delete X from R's disk.

4.4.2 Protocol 2

Protocol 2 (Figure 4) assumes a physically secure (PS) channel between R and P; e.g., P could be a TPM directly connected to R's computer. Thus no key agreement is required on this channel. Furthermore, Protocol 2 uses *asymmetric* encryption to secure time-limited data (independent of the PS channel).

In Figure 4 we use the notation from the beginning of this section. Additionally $\{M\}_{PK_R^+}$ (and $\{M\}_{PK_R^-}$) denote the public-key encryption (and decryption) of message M with the public (private) key PK_R^+ (PK_R^-) of principal R, respectively. Protocol 2 is based on R's authenticated broadcast, indicated by *, of the mid-term public keys PK_R^+. These public keys form part of freshly generated key pairs and are broadcasted along with their expiration times t_e. An example broadcast is the authenticated publication of PK_R^+ along with t_e on the receiver's website, or the receiver's reply to a request by the sender (not shown in Figure 4). The corresponding secret keys PK_R^- are not stored on R but on a porter directly connected to R over a PS channel. At any point in time, the sender may pick the public key that corresponds to the desired data lifetime, use it to encrypt the time-limited data, and transmit the message to the receiver, along with the respective expiration time (thereby enabling the receiver to identify the right secret key). The messages transmitted over the DY channel are authenticated.

4.4.3 Comparison

Protocols 1 and 2 differ in (i) how they create the mid-term encryption key and (ii) how they achieve forward-secrecy on the communication channel between R and P.

Protocol 1 uses key contributions by both the sender and the receiver to establish the symmetric encryption key and assumes a DY channel between R and P. This requires DH session key establishments on both communication channels. A typical application for Protocol 1 is the forward-secure email-communication of a company (under our full-compromise attacker model) using a trusted remote device for key management, e. g., a key card or other special-purpose devices.

In contrast to this, Protocol 2 uses asymmetric encryption with key pairs created by the receiver. The public keys may, e. g., be announced on a private user's webpage. Protocol 2 does not require DH key establishment on the communication channel between S and R. Due to the PS channel, it also does not require DH key establishment between R and P. The communication devices using Protocol 2 must be able to perform public-key operations; for example, the porter can be a TPM attached to R. In a slightly dif-

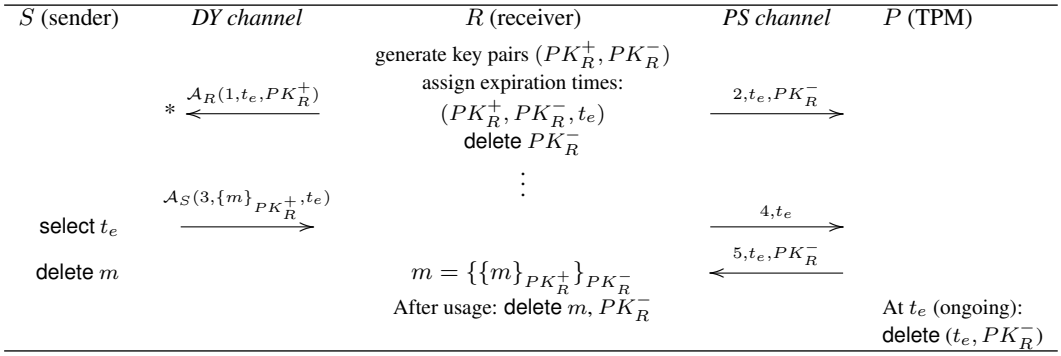


Figure 4: Protocol 2. The protocol assumes a physically secure (PS) channel between R and P (e. g., P is an HSM physically wired to R's hard disk). Hence, messages between S and R need not to be further protected by encryption or authentication measures. The mid-term key pair used for data encryption and decryption is (PK_R^+, PK_R^-). There are no ephemeral keys used.

ferent setting, this protocol can also be applied if S and R are porter devices that can directly communicate. In this case, the operations on the PS channel are simple storage and data retrieval operations to and from the memory of the porter.

In summary, if the communication devices can perform key management, they can also be used for key storage; if not, the key management should be outsourced to a suitable porter. We also note that the building blocks of Protocols 1 and 2 can be mixed, e. g., one can build an implementation that uses symmetric encryption while relying on a TPM connected by a PS channel.

4.5 Formal Protocol Analysis

We now construct formal models of our protocols and analyze the secrecy of the message m with respect to our attacker model (Section 2.3) using the Scyther tool [15]. We chose Scyther since it provides support for revealing the principals' states and enables us to analyze forward secrecy under full compromise [6, 7]. We first provide background information on Scyther.

4.5.1 Background on Scyther

Scyther is a tool for the symbolic automatic analysis of the security properties of cryptographic protocols (typically confidentiality or variants of authenticity). It assumes perfect cryptography, meaning that an attacker gains no information from an encrypted message unless she knows the decryption key (this is a standard assumption in symbolic methods). Scyther takes as input a role-based description of a protocol in which the intended security properties are specified using *claims*. Claims are of the form claim(Principal,Claim,Parameter), where Principal is the user's name, Claim is a security property (such as 'secret'), and Parameter is the term for which the security property is checked.

Recent versions of Scyther can analyze protocols with respect to a family of attacker models, ranging from a standard Dolev-Yao style network attacker to stronger attackers capable of various types of compromise. The attacker model is specified by selecting a set of attacker capabilities, such as revealing the short-term or long-term secrets of users. To analyze our protocols, we enable the following attacker capabilities: (i) Long-term key reveal for all principals and for other parties *after* the protocol execution, (ii) Session (short-term) key reveal for all parties not part of the current protocol execution, and (iii) Session-state reveal, which reveals the entire contents of the session-state of the parties. Together, these capabilities model the attacker from Section 2.3.

For most protocols and properties, the tool either finds an attack or establishes the unbounded verification of the protocol's properties with respect to the specified attacker model. In the remaining cases, bounded verification is performed where the *bound* defines the number of considered runs, i. e., the maximum number of parallel threads (or executions of role descriptions) executed by honest principals. This bounded result is similar to model-checking approaches for formal protocol verification. Attacks such as replay or man-in-the-middle attacks are typically found within the bound of two or three runs for many protocols (e. g., [5])[5]. The verification of over 100 protocols in [16] showed that no attacks were found that involved more runs than the number of principals in the protocol (except for protocols specifically constructed as counterexamples).

4.5.2 Analysis of Protocol 1

We model Protocol 1 (Figure 3) using eight send and receive events for the three principals S, R, and P. The complete protocol models and the tool itself are available at [2]. To give some intuition, we display the part that models the sender S:

```
role S {
  const rS: Nonce;           // S's DH key contribution
  const te: Nonce;           // expiration time
  const M:  Nonce;           // time-limited data
  var beta: Ticket;          // R's DH key contribution

  claim_sidS(S, SID, te);    // mark T_e as session id
  // Phase 1
  send_1(S,R,g1(rS),te,{l1,g1(rS),te}sk(S));
  recv_5(R,S,g1(rS),beta,te,{l5,g1(rS),beta,te}sk(R));
  // Phase 2
  send_!6(S,R,{l6a,M}g2(beta,rS),te,
    {l6b,{l6a,M}g2(beta,rS),te}sk({S});
  claim_s(S, Secret, M);
}
```

When using Scyther, security properties are modeled as local properties: If an agent executes a particular role, what can be concluded about the state of other agents or the attacker's knowledge? Here we analyze whether the protocol ensures the secrecy of m after the execution of an instance of S or R, and the secrecy of K after the execution of an instance of P, both under full compromise. In particular, we verified the following claims: S: claim(S,Secret,m), R: claim(R,Secret,m), and R: claim(P,Secret,K). As Scyther currently does not support explicit key expiration times, we model the expiration as happening immediately after the protocol execution, i. e.,

[5]The security analysis in [5] indicates that the Ephemerizer protocol is secure in terms of secrecy but insecure regarding integrity. The analysis is based on two (or three) runs.

Figure 5: Scyther result for Protocol 1.

after the key is retrieved from P. This is a worst case model because early key expiration only gives the attacker more knowledge at earlier times and thus more possibilities for attacks.

Figure 5 shows the results of the Scyther analysis. Scyther validates that no attacks exist on the model of Protocol 1 that involve less than four honest agent runs. For bounds of four or more parallel runs, the verification process did not terminate (within a day) due to the complexity of the analysis. Similar to bounded model-checking this neither falsifies the protocol nor proves its correctness, but establishes that no attacks exist within the given bounds.

4.5.3 Analysis of Protocol 2

We model Protocol 2 (Figure 4) in Scyther by two send events over the DY channel. Messages over the physically secure channel are not modeled as events because they are not subject to compromise (as opposed to the DY channel). Consequently, we verified the following two claims: S: claim(S,Secret,m) and R: claim(R,Secret, m). The input file provided to the Scyther tool can be found at [2]. The automatic analysis validates that no attacks exist on the model of Protocol 2 that involve less than ten honest agent runs. For bounds of ten or more parallel runs, the verification process did not terminate (within a day).

5. IMPLEMENTATION AND EVALUATION

We now examine possible realizations of key storage devices (Section 5.1), describe results from our mobile phone prototype implementation (5.2), and evaluate our solution (5.3).

5.1 Possible Realizations of Porter Devices

Dedicated Platforms. One possible realization of the porter device uses a platform that is embedded in the receiver device or is (occasionally) attached to the receiver. Example platforms on which porter functionality (i. e., key storage and delayed deletion) can be implemented are dedicated hardware security modules ([1] is an example of a platform that offer porter functionality). Note that our solution does not assume that the porter is tamper-resistant. Existing TPM modules could be used as porters but their functionality would have to be extended with delayed key deletion. TPMs used for this purpose must have an internal battery and clock; these are typically available in more advanced platforms (e. g., the IBM 4758 Cryptographic Coprocessor [14]). Typically, the communication with such a dedicated platform is a physically secure channel (a wired link); thus no additional measures are needed to guarantee the forward secrecy of the communication to and from the porter.

Mobile Phones. Private users might not have access to dedicated platforms. However access to mobile phones is widespread, so they are natural candidates for porter devices. For most users, their primary mobile phone is always operational and users pay close attention to their correct functioning and charging. The usability of mobile phones and personal computers has improved over the years and there are many convenient (wireless and wired) channels through which these devices can communicate and synchronize (e.g., Bluetooth [10]). The storage requirements of our solution are easily met with today's smartphone platforms (see Section 5.2). The communication between a mobile phone porter and the user's device must be forward-secret. Message secrecy is preserved, even given mobile phone loss prior to key deletion, as discussed in Section 4.3.

5.2 Prototype Implementation

To demonstrate the practical feasibility of our solution, we developed a prototype implementation of Protocol 1 for the communication between the receiver and the porter device. Our porter is a NexusOne [35] mobile phone (firmware 2.1, kernel 2.6.29, Android OS, 512 MB memory), depicted in Figure 6a. The receiver is implemented on a laptop running MacOSX 10.6.2.

The communication between the receiver and the porter is based on Bluetooth, using the Bluecove library [9], which we recompiled for a 64-bit Mac. Cryptographic operations are implemented using the Bouncycastle [4] library. To secure the key K on the wireless channel, we use symmetric AES/CBC/PKCS5 encryption with a 256-bit key that is derived from the established ephemeral DH key L (or L') by a SHA-256 hash. All messages are authenticated by a 32-byte HMAC-SHA with a 224-bit key that is pre-shared between the laptop and the mobile phone. Furthermore, we use 2-byte packet IDs, 8-byte timestamps, a 16-byte initialization vector for AES, and 1024-bit DH-key pairs. We use base64 encoding to transmit the packets over an RFComm Bluetooth connection and the tool `srm` to perform secure deletions on the laptop.

Figure 6b displays the execution times and the standard deviation for 100 runs as measured on the laptop. We see that key storage and retrieval are below one second, once the Bluetooth connection has been established. The times for key storage have a larger variance than for key retrieval since files are created and written rather than just read.

The key storage and delayed deletion functionality on the porter is implemented as follows. We store the keys along with their expiration times in files. To ensure the timely deletion of a file (i. e., key), we set an alarm service to automatically trigger the deletion of the respective file upon the timestamp's expiration. If the phone is shut down and rebooted before the expiration time, a background process (triggered by the boot-complete system broadcast) parses the key files, deletes files with expired timestamps, and resets the alarms. Figure 6c displays the time for setting the system alarm for different numbers of keys (files). We see the linear dependency of the number of keys on the alarm reset time. This background process does not significantly degrade the usability of the device.

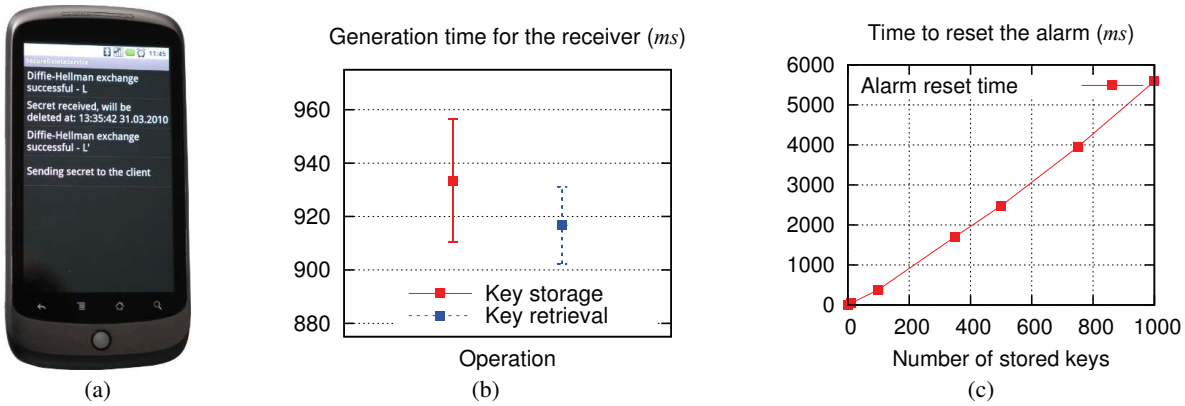
Given the execution times in Figure 6 and a consumption of 1.5 kB for program storage and 0.18 kB per key, our prototype implementation confirms the usability of our approach in practice.

Note on Secure Deletion

When exploring implementation options, we observed that many embedded devices have limited functionality for secure deletion due to OS characteristics (like versioning) or hardware specifics (e.g., NAND storage often uses log-structured sequential writes). Enabling secure deletion on these devices is subject to recent research, examples include [26] for the Android YAFFS file system, [30] for versioning file systems, and [34] regarding data remanence in flash memory devices. Secure deletion may not be realized on certain off-the-shelf devices and care should thus be taken in the selection of the porter device.

5.3 Integration with Applications

Given a functional porter device, our solution can be integrated



(a) (b) (c)

Figure 6: (a) Prototype implementation of the key storage (porter) functionality on a NexusOne mobile phone. (b) Protocol execution times for the receiver. The plot shows the average times and the standard deviation for 100 runs of Protocol 1. (c) Time for resetting the system alarm (used to delete keys at their expiration time) after a phone reboot for different numbers of keys.

in many applications, such as file storage, web services, and e-mail.

File Storage. The simplest application of our solutions is to local file storage, where a device locally stores confidential data that should be inaccessible after the expiration time. To enable this, the device encrypts the data locally with a key stored on a porter device. Here, our protocols can be substantially simplified: the only communication that needs to be forward-secret is that between the device and the porter (this can be further simplified if this channel is physically secure). Remote file storage is similar to local file storage in that the only device that has access to the data decryption keys is the device that created them. However, the communication between the user's device and the remote file server where the data is stored must be forward-secret.

Web Services. Another application is where users share their data (e. g., pictures, movies, files) using remote storage in the form of a web service. The data is to be kept secret even in the case of full device (sender and receiver) and service compromise. In contrast to standard remote file storage, here the communication key must be agreed upon between the sender (who uploads the data) and the receiver(s) (who download the data). The receiver stores the key on a porter device and obtains it when it downloads the data. The communication used for key agreement must be forward secret.

E-mail. Finally, we consider the scenario where the sender and the receiver wish to preserve the secrecy of their e-mail correspondence. In this scenario, the parties first agree on the keys that they will use for their communication and on their expiration times. They then store the keys on their respective porter devices and exchange e-mail. Although we could directly use Protocol 1, it can be optimized by making a mobile phone porter establish the keys directly with the sender. This can be alternatively done via e-mail exchange, without the participation of the receiver. The receiver could be notified that the keys are established when it synchronizes (e. g., over an IMAP server) with the e-mail communication.

6. RELATED WORK

Shoup [32] defined three notions for the compromise of principals: *static corruptions* (in which principals are either compromised or not), *adaptive corruptions* (in which long-term keys may be compromised), and *strong adaptive corruptions* (in which the compromise of principals reveals both long-term and short-term stored secrets). In our attacker model, we build on the third notion by considering full device and user password compromise after a specific time.

Methods for protecting data confidentiality under device compromise include secret sharing [31], threshold cryptography [17], and forward secrecy [22]; we focus on the last method. Canetti et al. [13] proposed a forward-secure public-key encryption scheme in which a receiver evolves its private key such that it can only decrypt messages with a later timestamp. A similar idea was adopted by Bellare et al. [8] for forward-secure digital signature schemes. We cannot use such approaches because private keys cannot evolve when devices are inactive.

The confidentiality of data exchanged between individuals or organizations is attracting increasing attention. Centralized systems such as [20] for server-based sharing and storage of personal data offer access control and data deletion at user-defined or automatically derived times. However, they require full trust in the service provider to treat passwords and data confidentially and to delete both when specified. Ephemerizer-based solutions [25, 28, 29] similarly require trust in a central server. As discussed in [21], this does not ensure the data confidentiality in the presence of service-provider mismanagement and legal action to reveal data.

7. CONCLUSION

We addressed the problem of data confidentiality in scenarios where attackers can observe the communication between principals and can also fully compromise the principals after the data has been exchanged, thereby revealing the entire state of the principals' devices. We explored the design space of solutions to this problem and proposed two protocols that use key storage on porter devices along with explicit deletion and forward secret subprotocols to achieve secrecy under full device, user and communication compromise. The solutions provide users with full control over their data privacy. We formalized our proposed solutions and analyzed them using an automatic verification tool. Our prototype implementation shows their practicality and feasibility.

Acknowledgment

The authors thank Claudio Marforio for his work on the implementation of the prototype.

8. REFERENCES

[1] Privat Server HSM (Hardware Security Module). `http://www.arx.com/products/hsm.php`.

[2] Scyther protocol models for keeping data secret under full compromise using porter devices. http://people.inf.ethz.ch/cremersc/scyther/DataDeletion, Oct 2010.

[3] Chris Alexander and Ian Goldberg. Improved user authentication in off-the-record messaging. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES)*, pages 41–47, New York, 2007. ACM.

[4] Bouncy Castle Crypto APIs. http://www.bouncycastle.org.

[5] Charu Arora and Mathieu Turuani. Validating integrity for the Ephemerizer's protocol with CL-Atse. In *Formal to Practical Security: Papers Issued from the 2005-2008 French-Japanese Collaboration*, pages 21–32. Berlin, 2009.

[6] David Basin and Cas Cremers. Degrees of security: Protocol guarantees in the face of compromising adversaries. In *Proceedings of the 24th International Workshop on Computer Science Logic (CSL)*, pages 1–18. Springer, 2010.

[7] David Basin and Cas Cremers. Modeling and analyzing security in the presence of compromising adversaries. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, pages 340–356. Springer, 2010.

[8] Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, pages 431–448, London, 1999.

[9] Bluecove. Java library for Bluetooth (JSR-82 implementation). http://bluecove.org.

[10] Bluetooth SIG, Inc. Bluetooth specification version 3.0 + HS, 2009.

[11] Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-record communication, or, why not to use PGP. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES)*, pages 77–84, New York, 2004. ACM.

[12] Colin Boyd and Anish Mathuria. *Protocols for Authentication and Key Establishment*. Springer, 2003.

[13] Ran Canetti, Shai Halevi, and Jonathan Katz. A forward-secure public-key encryption scheme. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 255–271. Springer, 2003.

[14] IBM Corporation. IBM PCI Cryptographic Coprocessor. General Information Manual. http://www-03.ibm.com/security/cryptocards.

[15] Cas Cremers. Scyther. A tool for the automatic verification of security protocols. http://people.inf.ethz.ch/cremersc/scyther.

[16] Cas Cremers. *Scyther—Semantics and Verification of Security Protocols*. PhD thesis, Eindhoven University of Technology, 2006.

[17] Yvo Desmedt and Yair Frankel. Threshold cryptosystems. In *Proceedings on Advances in Cryptology (CRYPTO)*, pages 307–315, New York, 1989. Springer.

[18] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, 1992.

[19] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[20] Drop. Simple real-time sharing, collaboration, presentation. http://drop.io.

[21] Roxana Geambasu, Tadayoshi Kohno, Amit Levy, and Henry M. Levy. Vanish: Increasing data privacy with self-destructing data. In *Proceedings of the 18th USENIX Security Symposium*, pages 299–315. USENIX Association, 2009.

[22] Christoph G. Günther. An identity-based key-exchange protocol. In *Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques on Advances in Cryptology (EUROCRYPT)*, pages 29–37, New York, 1990. Springer.

[23] Prateek Gupta and Vitaly Shmatikov. Key confirmation and adaptive corruptions in the protocol security logic. In *Proceedings of the Joint Workshop on Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis (FCS-ARSPA)*, 2006.

[24] Peter Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the 6th USENIX Security Symposium (SSYM), Focusing on Applications of Cryptography*, pages 77–90, Berkeley, California, 1996. USENIX Association.

[25] Disappearing Inc. http://www.disappearing-inc.com.

[26] Jaeheung Lee, Junyoung Heo, Yookun Cho, Jiman Hong, and Sung Y. Shin. Secure deletion for NAND flash file system. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 1710–1714, 2008.

[27] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[28] Radia Perlman. The Ephemerizer: Making data disappear. *Journal of Information System Security*, 1:51–68, 2005.

[29] Radia Perlman. File system design with assured delete. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. ISOC, 2007.

[30] Zachary N. J. Peterson, Randal Burns, Joe Herring, Adam Stubblefield, and Aviel D. Rubin. Secure deletion for a versioning file system. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)*, pages 143–154, Berkeley, California, 2005. USENIX Association.

[31] Adi Shamir. How to share a secret. *Commununications of the ACM*, 22(11):612–613, 1979.

[32] Victor Shoup. On formal models for secure key exchange. Research Report RZ 3120, IBM Research, 1999.

[33] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Life or death at block-level. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 379–394, Berkeley, California, 2004. USENIX Association.

[34] Sergei Skorobogatov. Data remanence in flash memory devices. In *Proceedings of the Cryptographic Hardware and Embedded Systems Workshop (CHES)*, pages 339–353, 2005.

[35] NexusOne Smartphone. http://www.htc.com.

[36] Scott Wolchok, Owen S. Hofmann, Nadia Heninger, Edward W. Felten, J. Alex Halderman, Christopher J. Rossbach, Brent Waters, and Emmett Witchel. Defeating Vanish with low-cost Sybil attacks against large DHTs. In *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS)*. ISOC, 2010.

Familiarity Breeds Contempt

The Honeymoon Effect and the Role of Legacy Code in Zero-Day Vulnerabilities

Sandy Clark
University of Pennsylvania
saender@cis.upenn.edu

Stefan Frei
Secunia
sfrei@secunia.com

Matt Blaze
University of Pennsylvania
blaze@cis.upenn.edu

Jonathan Smith
University of Pennsylvania
jms@cis.upenn.edu

ABSTRACT

Work on security vulnerabilities in software has primarily focused on three points in the software life-cycle: (1) finding and removing software defects, (2) patching or hardening software after vulnerabilities have been discovered, and (3) measuring the rate of vulnerability exploitation. This paper examines an earlier period in the software vulnerability life-cycle, starting from the release date of a version through to the disclosure of the fourth vulnerability, with a particular focus on the time from release until the very first disclosed vulnerability.

Analysis of software vulnerability data, including up to a decade of data for several versions of the most popular operating systems, server applications and user applications (both open and closed source), shows that *properties extrinsic to the software play a much greater role in the rate of vulnerability discovery than do intrinsic properties such as software quality.* This leads us to the observation that (at least in the first phase of a product's existence), software vulnerabilities have different properties from software defects.

We show that the length of the period after the release of a software product (or version) and before the discovery of the first vulnerability (the 'Honeymoon' period) is primarily a function of familiarity with the system. In addition, we demonstrate that legacy code resulting from code re-use is a major contributor to both the rate of vulnerability discovery and the numbers of vulnerabilities found; this has significant implications for software engineering principles and practice.

1. INTRODUCTION

Software vulnerabilities are the root cause of many security breaches, so understanding software systems is essential to developing models for how and when to invest effort in securing software. The most important software systems to understand are those of large scale and those in wide use. Since almost all software systems today are large and complex, we can focus our attention on those in wide use. Ranging from document preparation programs to web browsers and operating systems, such systems can each comprise millions of lines of source code, a very rough measure of software complexity. Given the importance of such systems, models for their creation, use, maintenance and upgrades - their "life-cycle" - are clearly necessary.

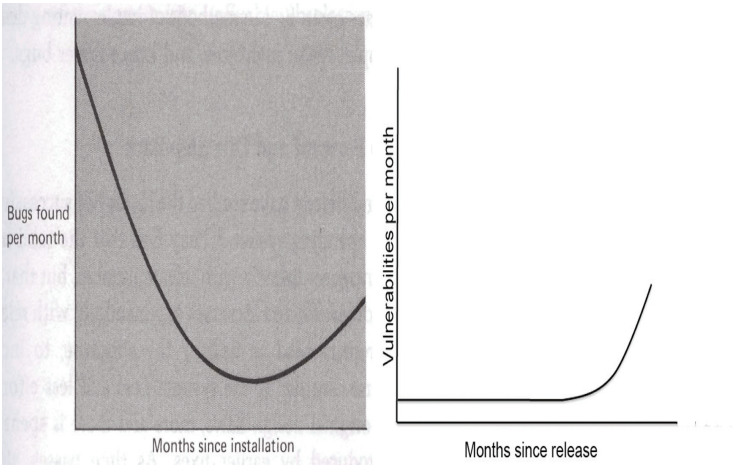


Figure 1: Bugs per month, Left:Figure 11.2 from "The Mythical Man Month", Right: Security vulnerabilities per month

Models are useful in estimating project costs and timing. For example, if a model predicts that the bug discovery rate drops rapidly after an initial flurry of discoveries, this fact can be used to determine when software is ready for release: once the rate has reached an acceptable level, the software can be shipped. Such estimation can have significant economic effects upon an enterprise: ship too early and pay a price in service calls; ship too late and potentially lose customers who might look elsewhere. A powerful predictive model can therefore be worth significant amounts of revenue, as it allows trading development costs and time against a combination of sales revenue and maintenance costs.

Software Reliability Models (SRMs) are primarily concerned with increasing the quality of the code by predicting and locating software defects. A major assumption made

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

by SRMs is that software is released with some number of defects that can be categorized based on how easy each is to find. A further assumption is made that the easy-to-find defects are discovered and fixed *early* in the software life-cycle, quickly leading to a state where only difficult-to-find vulnerabilities are left and the software can be considered reliable. Figure 11.2 from Brooks [5] is reproduced on the left of Figure 1 to illustrate this point.

Software Vulnerability Discovery Models (VDMs) resemble SRMs, but VDMs focus predominantly on predicting attacks against mature software systems. VDMs rely on the *intrinsic qualities* of the software for a measure of its initial security. For a VDM the expectation is that the low-hanging fruit vulnerabilities are found quickly and patched. The remaining vulnerabilities (which are increasingly difficult to find) are presumed to take much longer to discover, and the software is considered "secure". A VDM with those expectations would predict that vulnerabilities are found fastest shortly after the release of a product, and the rate of discovery decreases thereafter.

The implications of such a VDM are significant for software security. It would suggest, for example, that once the rate of vulnerability discovery was sufficiently small, that the software is "safe" and needs little attention. It also suggests that software modules or components that have stood this "test of time" are appropriate candidates for reuse in other software systems. If this VDM model is wrong, these implications will be false and may have undesirable consequences for software security.

Unlike much of the previous work [25, 2, 28] which focused on understanding time to exploit after a vulnerability has been discovered, this paper focuses on measuring time to vulnerability discovery.

The remainder of the paper is organized as follows. Section 2 describes our unique dataset of vulnerabilities, covering several versions of the most popular software products, operating systems, server applications and user applications.

In Section 3 we analyze this data, which show that the period between the release date of a product and its very first 0-day vulnerability is considerably longer than the mean time between the first vulnerability and second or between the second and the third. We call this unexpected grace period the *honeymoon effect* and believe it to be important, because these numbers challenge our expectations and intuition about the effect of software quality on security. The interval between software release and the discovery of it's first 0-day vulnerability also appears to be a strong predictor of the arrival rate of subsequent vulnerability discoveries.

The honeymoon effect also illustrates another incompatibility between current software engineering practices and security: the effect of code reuse. "Good programmers write code, Great programmers reuse " is a well-known aphorism, and the assumption made is that reusing code is not only more efficient, but since the code has already been deployed successfully, it is more reliable and therefore, by implication, also more secure. In Section 4 our data again show this is not the case.

We set our results in the context of prior work in Section 5 and conclude the paper by summarizing our claims and discussing the implications for engineering secure software systems in Section 6.

2. OUR DATASET

In this paper, we are concerned specifically with the early post-release vulnerability life-cycle for modern, mass market software, including operating systems, web clients and servers, text and graphics processors, server software, and so on.

Our analysis focuses on publicly distributed software released between 1999 and 2007. (2007 is the latest date for which complete vulnerability information was reliably available from various published data sources). We included both open and closed source software.

To encompass the most comprehensive possible range of relevant software releases, we collected data about all released versions of the major operating systems (Windows, OS X, Redhat Linux, Solaris, FreeBSD), all released versions of the major web browsers (Internet Explorer, Firefox, Safari), and all released versions of various server and end user applications, both open and closed source. The server and user applications were based on the top 25 downloaded / purchased / favorite application identified in lists published by ZDNet, CNet, and Amazon, excluding only those applications for which accurate release date information was unavailable or that were not included in the vulnerability data sources described below. In total, we were able to compile data about 38 of the most popular and important software packages.

For each software package and version during the period of our study, we examined public databases, product announcements, and published press releases to assign each version a release date. For the period of versions (1990-2007) and for the period of vulnerabilities (1999-2008), we identified 700 distinct released versions ('major' and 'minor') of the 38 different software packages.

We then compiled a dataset of more than 30,000 exploitable vulnerabilities that were disclosed during the period under analysis (January 1999 through January 2008). Our baseline sources were publicly available databases from the National Vulnerability Database (NVD) [23] and from the Common Vulnerabilities and Exposures (CVE) [9] initiative that feeds NVD. (For each vulnerability, NVD provides a publication date, a short description, a risk rating, references to original sources, and information on the vendor, version and name of the product affected.) We also downloaded, parsed, and correlated the information from over 200,000 individual security bulletins from several "Security Information Providers" (SIPs), choosing the set of SIPs based on criteria such as independence, accessibility, and available history of information. Ultimately, we processed all security advisories from the following seven SIPs: Secunia, US-CERT, SecurityFocus, IBM ISS X-Force, SecurityTracker, iDefense's (VPC), and TippingPoint(ZDI) [29, 33, 30, 14, 34, 31, 15, 32].

For this study, we selected from these bulletins and database entries bugs identified as *exploitable vulnerabilities* that render the software vulnerable to actual attack and for which a practical exploit has been demonstrated. We then calculated the *initial disclosure date* for each exploitable vulnerability to be the earliest calendar day on which information on a specific vulnerability is made freely available to the public in a consistent format by some recognized published source [11]. To help ensure accuracy, we manually checked and corrected over 3,000 instances of software version information for the specific product versions under analysis in this paper to normalize for inconsistencies in NVD's vulnerability to product mapping.

3. THE HONEYMOON EFFECT

Virtually all mass-market software systems undergo a lengthy period, after their release, during which end-users discover and report bugs and other deficiencies. Most software suppliers (whether closed-source or open-source) build into their life-cycle planning a mechanism for reacting to bug reports, repairing defects, and releasing patched versions at regular intervals. The number of latent bugs in a particular version of a given version of a given piece of software thus tends to decrease over time, with the initial, unpatched, release suffering from the largest number of defects. (This excludes, of course, defects introduced by patches, which are a minority in practice). In systems where bugs are fixed in response to user reports, the most serious and easily triggered bugs would be expected to be reported early, with increasingly esoteric defects accounting for a greater fraction of bug reports as time goes on.

Empirical studies in both the classic [5] and the current [16] software engineering literature have shown that, indeed, this intuition reflects the software life-cycle well (see Figure 2). Invariably, these and other software engineering studies have shown that the rate of bug discovery is at its highest immediately after software release, with the rate (measured either as inter-arrival time of bug reports or as number of bugs per interval) slowing over time.

Note that some (but not all) of the bugs discovered and repaired in this process represent *security vulnerabilities;* in security parlance a vulnerability that allows an attacker to exploit a newly discovered, previously unknown bug is called a *0-day* vulnerability. Virtually all software vendors give high priority to repairing defects once a 0-day exploit is discovered.

It seems reasonable, then, to presume that users of software are at their most vulnerable, with software suffering from the most serious latent vulnerabilities, immediately after a new release. That is, we would expect attackers (and legitimate security researchers) who are looking for bugs to exploit to have the easiest time of it early in the life cycle. This, after all, is when the software is most intrinsically weak, with the highest density of "low hanging fruit" bugs still unpatched and vulnerable to attack. As time goes on, after all, the number of undiscovered bugs will only go down, and those that remain will presumably require increasing effort to find and exploit.

In other words, to the extent that security vulnerabilities are a consequence of software bugs, conventional software engineering wisdom tells us to expect the discovery of 0-day exploits to follow the same pattern as other reported bugs. The pace of exploit discovery should be at its most rapid early on, and slowing down as the software quality improves and the "easiest" vulnerabilities are repaired.

But our analysis of the rate of the discovery of exploitable bugs in widely-used commercial and open-source software, tells a very different story than what the conventional software engineering wisdom leads us to expect. In fact, new software overwhelmingly enjoys a *honeymoon* from attack for a period after it is released. The time between release and the first 0-day vulnerability in a given software release tends to be markedly longer than the interval between the first and the second vulnerability discovered, which in turn tends to be longer than the time between the second and the third. That is, when the software it at its *weakest*, with the "easiest" exploitable vulnerabilities still unpatched, there is a *lower* risk that this will be discovered by an actual attacker on a given day than there will be *after the vulnerability is fixed!*

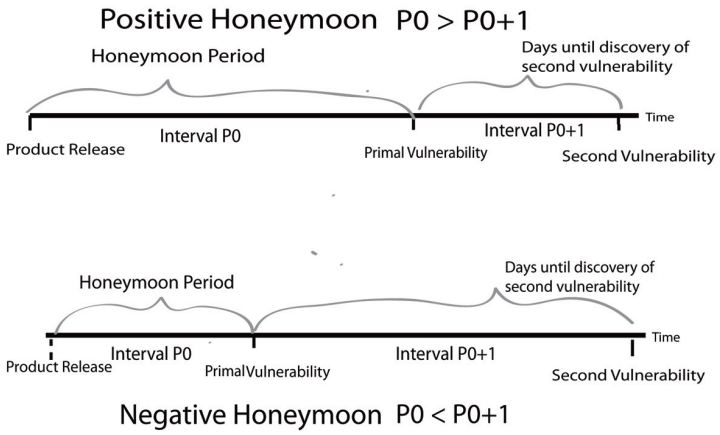


Figure 3: The Honeymoon Period, both Positive and Negative time-lines

3.1 The Honeymoon Effect and Mass-Market Software

For the purposes of this paper, we define the first (publicly reported) exploitable vulnerability as the *primal* vulnerability, we define a software release as experiencing a *positive honeymoon* if the interval p_0 between the (public) release of the software and the primal vulnerability in the software is greater than the interval p_{0+1} between the primal vulnerability and the second(publicly reported) vulnerability.(see Figure 3) We will refer here to the interval p_0 as the *honeymoon period* and the ratio p_0/p_{0+1} as the *honeymoon ratio*. In other words, a software release has experienced a positive honeymoon when its honeymoon ratio > 1.

We examined 700 software releases of the most popular recent mass-market software packages for which release dates and vulnerability reports were available (see Section 2). In 431 of 700 (62%) of releases, the honeymoon effect was positive. Most notably, the median overall honeymoon ratio (including both positive and negative honeymoons) p_0/p_{0+1} was 1.54. That is, the median time from initial release and the primal vulnerability is 1 1/2 times greater than the time from primal to the discovery of the second. The honeymoon effect is not only present, it is quite pronounced, and the effect is even more pronounced when we exclude minor version updates and focus on major releases. For major releases, the honeymoon ratio(including both positive and negative honeymoons) rises to 1.8.

Remarkably, positive honeymoons occur across our entire dataset for all classes of software and across the entire period under analysis. The honeymoon effect is strong whether the software is open- or closed- source, whether it is an OS, web client, server, text processor, or something else, and regardless of the year in which the release occurred.(see Table 1)

Although the honeymoon effect is pervasive across the entire dataset, one factor appears to influence its length more than any other: the re-use of code from previous releases, which, counter-intuitively, *shortens* the honeymoon. Soft-

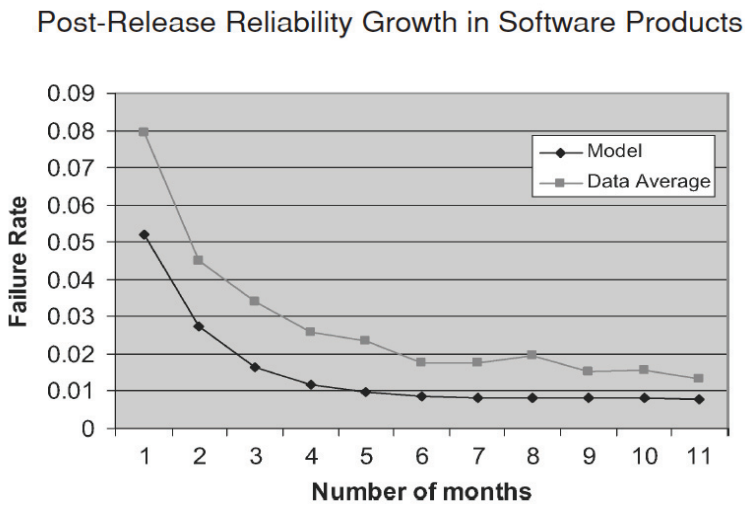


Figure 2: Current Software Engineering literature supports the Brooks life-cycle model - image taken from "Post-release reliability growth in software products", *ACM Trans. Softw. Eng Methodol. 2008* see references

Table 1: Percentages of Honeymoons by Year

Year	Honeymoons
1999	56%
2000	62%
2001	50%
2002	71%
2003	53%
2004	49%
2005	66%
2007	58%

ware releases based on "new" code have longer honeymoons than those that re-use old code. We discuss this in detail in the following sections.

3.2 Honeymoons in Different Software Environments

The number of days in the honeymoon period varies widely from software release to software release, and ranged from a single day to over three years in our dataset. The length of the honeymoon presumably varies due to many factors, including the intrinsic quality of the software and extrinsic factors such as attacker interest, familiarity with the system, and so on.

To "normalize" the length of the honeymoon for these factors to enable meaningful comparisons between different software packages, the honeymoon ratio – the ratio of the time between release and the discovery of the first exploit and the time between the discovery of the first and the second – may be more revealing, since time to the second vulnerability discovery occurs in exactly the same software.

The median number of days in the honeymoon period across all 700 releases in our dataset was 110. The median honeymoon ratio across all releases is 1.54.

The honeymoon ratio remained positive in virtually all software packages and types. The effect is weaker, but also occurred, between the primal and second and second and third reported vulnerabilities, depending on the particular software package.

Figure 4 shows the median honeymoon ratio (and the median ratios for the intervals between the second, third and fourth vulnerabilities) for each operating system in the dataset. Figure 5 shows the median honeymoon ratio of servers, and Figure 6 shows end-user applications.

3.3 Open vs. Closed Source

The honeymoon effect is strong in both open- and closed-source software, but it manifests itself somewhat differently.

Of the 38 software systems we analyzed, 13 are open-source and 25 are closed-source. But of the 700 software releases in our dataset 171 were for closed-source systems and 508 were for open source. Open-source packages in our dataset issued new release versions at a much more rapid rate than their closed source counterparts.

Table 2: Median Honeymoon Ratio for Open and Closed Source Code

Type	Honeymoon Days	Ratios
Open Source	115	1.23
Closed Source	98	1.48

Yet in spite of its more rapid pace of new releases, open source software releases enjoyed a significantly longer median honeymoon before the first publicly exploitable vulnerability was discovered: 115 days, vs. 98 days for closed-source releases.(see Table 2)

The median honeymoon ratio, however, is shorter in open-source than in closed. The median ratio for all open-source releases was 1.23, but for closed source it was 1.48. Figure 7 shows the median honeymoon ratios for various open-source systems, and Figure 8 shows the median ratios for closed-source systems.

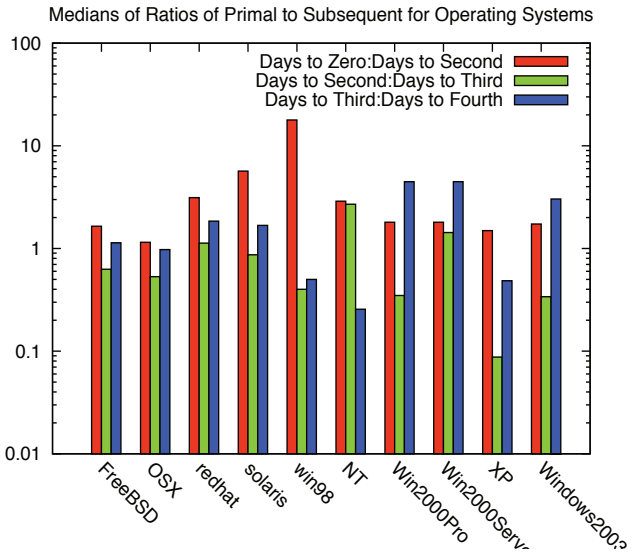


Figure 4: Honeymoon ratios of p_0/p_{0+1}, p_{0+1}/p_{0+2} and p_{0+2}/p_{0+3} for major operating systems. (Log scale. Note that a figure over 1.0 indicates a positive honeymoon).

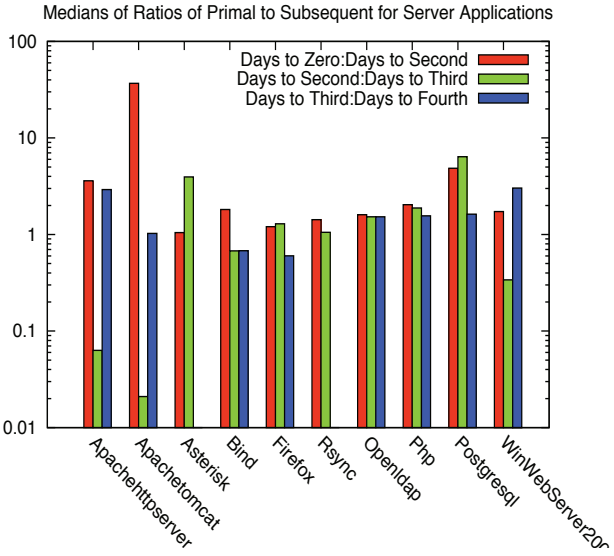


Figure 5: Honeymoon ratio of p_0/p_{0+1}, p_{0+1}/p_{0+2} and p_{0+2}/p_{0+3} for common server applications

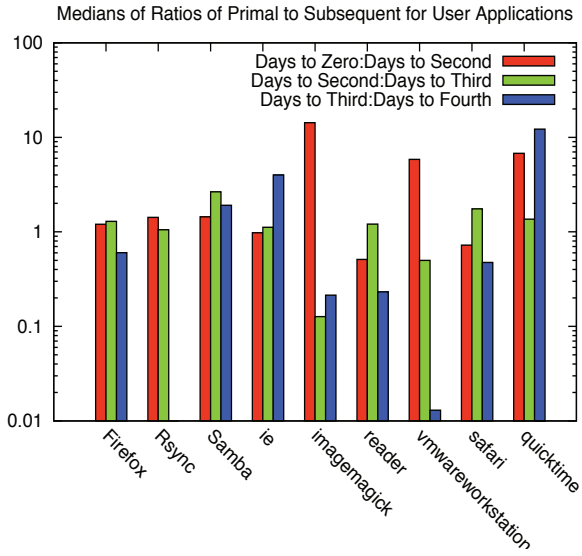


Figure 6: Honeymoon ratios of p_0/p_{0+1}, p_{0+1}/p_{0+2} and p_{0+2}/p_{0+3} for common user applications

The longer honeymoon period with a shorter honeymoon ratio for open-source software suggests that it not only takes longer for attackers to find the initial bugs in open-source software, but that the rate at which they "climb the learning curve" does not accelerate as much over time as it does in closed-source systems. This may be a surprising result, given that attackers do not have the opportunity to study the source code in closed-source systems, and suggests that familiarity with the system is related to properties *extrinsic to the system* and not simply access to source code.

4. THE HONEYMOON EFFECT AND PRIMAL VULNERABILITIES

To more fully understand the factors responsible for the honeymoon effect, we examined the attributes of a particular set of *primal* vulnerabilities. In this section we compare the honeymoon periods of this set and show that primal vulnerabilities are not a result of "low-hanging fruit", and that other extrinsic properties must apply.

It is well known that as complex software evolves from one version to the next, new features are added, old ones deprecated and changes are made, but throughout its evolution much of the standard code base of a piece of software remains the same. One reason for this is to maintain backward compatibility, but an even more prevalent reason is that code re-use is a primary principle of software engineering [18, 5].

In Milk or Wine [25] Ozment *et al* measured the portion of legacy code in several versions of OpenBSD and found that 61% of legacy (their term is 'foundational') code was still present 15 releases (and 7.5 years) later. This legacy code accounted for 62% of the total vulnerabilities found. While it is not possible to measure the amounts of legacy code from version to version in closed source products as it is for open source, it is well known that the major ven-

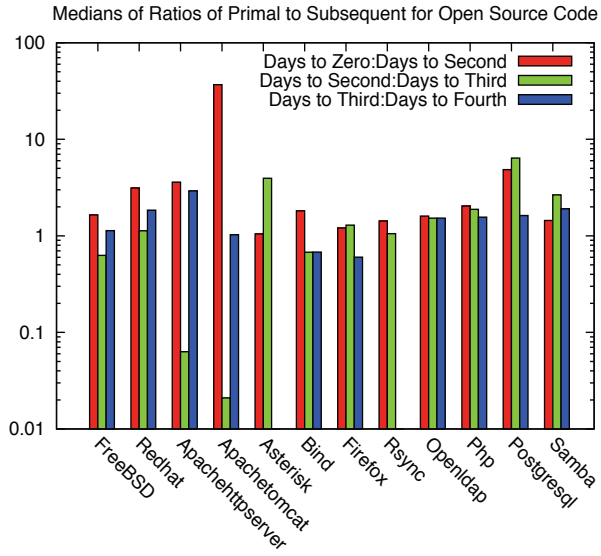


Figure 7: Ratios of p_0/p_{0+1} to p_{0+1}/p_{0+2} and p_{0+2}/p_{0+3} for open source applications

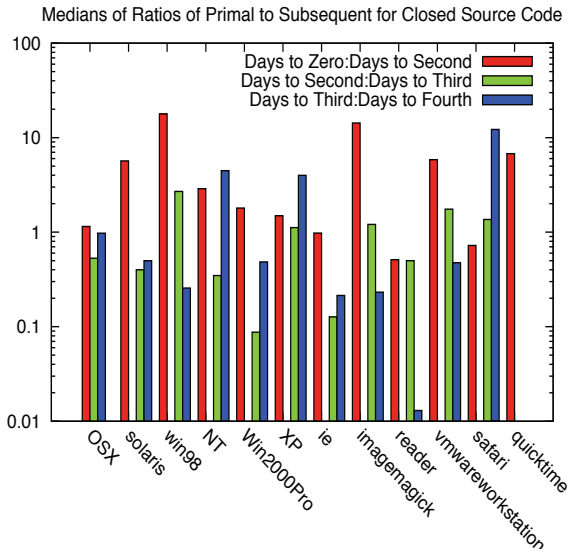


Figure 8: Ratios of p_0/p_{0+1} to p_{0+1}/p_{0+2} and p_{0+2}/p_{0+3} for closed source applications

dors strongly encourage code re-use among their collaborating developers [19], and more importantly, it *is* possible to measure the numbers of legacy vulnerabilities. By comparing the disclosure date of a vulnerability with the release dates and product version affected, it is possible to determine which vulnerabilities discovered in the current release result from earlier versions. For example, if a vulnerability V affects versions (k,...N) (0<k<N) of a product, but not versions (1,...,k-1) and was disclosed *after* the release date of version N, we know that the vulnerability was introduced into the product with version k, and that it stayed hidden until its discovery after the release of version N. We call these *regressive* vulnerabilities as they are those vulnerabilities which are not found through normal regression testing and may lie dormant through more than one version release(sometimes for years).[1] For the purposes of this paper, we define a *regressive* vulnerability as a primal vulnerability that was discovered to affect not only version N in which it was found, but also affect one or more earlier versions (versions N-1, N-2,...,1.0)

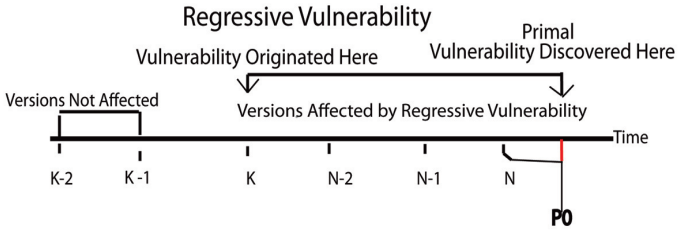


Figure 9: Regressive Vulnerability timeline

On the other hand, a *progressive* vulnerability is primal vulnerability which is discovered in version N and does not affect version N-1 or any earlier versions. A progressive vulnerability indicates that the vulnerability was introduced with the new version N. (see Figure 9)

Figure 10 shows that legacy vulnerabilities[2] make up a significant percentage of vulnerabilities across all products, e.g. 61% of the Windows Vista vulnerabilities originate in earlier versions of the OS, 40% of which originate in Windows 2000 released seven years earlier. This analysis shows that vulnerabilities are typically long-lived and can survive over many years and many product versions until discovered.

In order to ascertain whether regressive vulnerabilities could be the result of code reuse rather than configuration or implementation errors, we manually checked the NVD database description and the original disclosure sources for information regarding the type of vulnerability. We found that 92% of the regressive vulnerabilities were the result of code errors (buffer overflows, input validation errors, exception handling errors) which strongly indicates that a vulnerability that affects more than one version of a product is most likely a result of legacy code shared between versions. We removed the vulnerabilities which are the result of implementation or configuration errors from our dataset and focused exclusively on code errors.

4.1 Regressive Vulnerabilities

[1]In OpenBSD, Ozment *et al* states "It took more than two and a half years for the first half of these ... vulnerabilities to be reported." [25].

[2]including both regressives and progressives

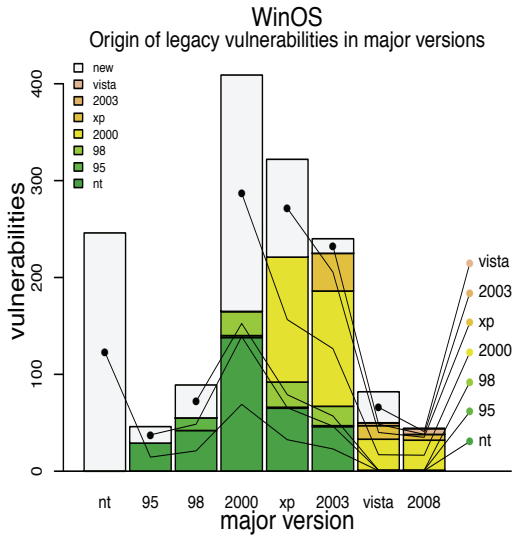


Figure 10: Proportion of legacy vulnerabilities in Windows OS

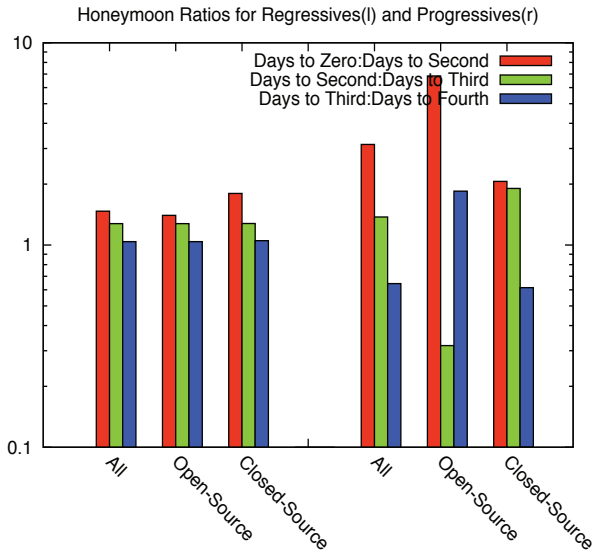


Figure 11: Honeymoon ratios of p_0/p_{0+1}, p_{0+1}/p_{0+2} and p_{0+2}/p_{0+3} for common user applications

If code reuse and an attacker's familiarity with the system has an effect on the rate of vulnerability discovery, then when one examines the *primal* vulnerabilities, one should expect to see that regressive vulnerabilities make up a significant percentage of them. And indeed, after examining all the primal vulnerabilities in our data set, we find that 77% of them are regressive. (ie, 77% of the primals were found to also affect earlier versions). Table 3 lists the percentages of regressives for all, open source, closed source primals. Table 3 also shows that the percentage of regressives is even higher for open source primals (rising up to 83%), and lower for closed source (59%). The high percentage of regressive vulnerabilities is surprising, because it shows that the majority of primal vulnerabilities, (the first vulnerability found after a product is released), are not the easy to find "low-hanging fruit" one would expect from conventional software engineering defects, instead these regressives lay dormant throughout the life-time of their originating release (and possibly several subsequent releases). If these regressives had been easy to find, then presumably, *they would have been found in the version in which they originated.*

Table 3: Percentages of Regressives and Regressive Honeymoons for all Primal Vulnerabilities

Type	Total Regressives	Total Regr. Honeymoons
ALL	77%	62%
Open Source	83%	62%
Closed Source	59%	66%

4.2 The Honeymoon Effect and Regressive Vulnerabilities

Another unexpected finding is that regressive vulnerabilities also experience the honeymoon effect. Because regressive vulnerabilities have been lying dormant in the code for more than one release, and because the attackers have had more time to familiarize themselves with the product, it seems reasonable to presume that the first of these vulnerabilities would be found in a shorter amount of time than time to find the second vulnerability (whether regressive or progressive). But, our analysis shows this isn't the case. The second column of Table 3 lists the percentages of regressives that were also honeymoons. In each case whether we looked at all regressives combined, only open source or only closed source, the percentages of honeymoons is in the low to mid 60th percentile - almost the same as the total honeymoon effect for all regressives and progressives combined. Closed source does exhibit a slightly longer honeymoon effect, but not significantly so. The existence of regressive honeymoons, especially in such high proportions indicates that properties extrinsic to the quality of the code, in particular an attacker's familiarity with the system play a much greater role early on in the life-cycle of a release than previously expected.

4.3 Regressives vs. Progressives

The strong presence of the honeymoon effect even among regressive vulnerabilities leads us to wonder what if any effect regressives might have on the length of the honeymoon period. Yes, regressive vulnerabilities experience a honeymoon, but is it longer or shorter than the honeymoon for progressive vulnerabilities? The honeymoon ratio provides insight into the length of the honeymoon period. Figure 11 shows the median honeymoon ratios for regressives (all, open and closed), progressives (all, open and closed), for the vulnerabilities p_0/p_{0+1}, through p_{0+2}/p_{0+3}. The median honeymoon ratio for regressive vulnerabilities is lower than that for progressives. In fact, the honeymoon ratio for progressive vulnerabilities is almost twice as long. This strongly suggests that familiarity with the system is a major con-

tributor to the time to first vulnerability discovery. Interestingly, it doesn't seem to have a significant effect on open source code, but closed source does seem to have a longer honeymoon period, even for regressives. In other words, *familiarity shortens the honeymoon.*

4.4 Less than Zero Days

Table 4: Percentages of Primals that are Less-than-Zero (released vulnerable to an already existing exploit) and the new expected median time to first exploit, for all products, Open source and Closed Source

Type	Percentages	Median Honeymoon Period
ALL	21%	83
Open Source	18%	89
Closed Source	34%	60

Dormant vulnerabilities are not the only cause of 0-days. Legacy vulnerabilities result in a second category of regressive 0-days for which there can be no honeymoon period. These *Less-than-Zero* days occur when a new version of a product is released vulnerable to a previously disclosed vulnerability. For example, the day Windows 7 was officially released, it was discovered that it was vulnerable to several current prominent viruses which had originally been crafted for Windows XP [35] Our research shows that less-than-zero days account for approximately 21% of the total legacy vulnerabilities found, with closed source code containing the most (34%)(see Table 4). In all cases the median number of days to first exploit is reduced by approximately 1/3 and the median honeymoon ratio drops from 1.54 to 1.0. From this we conclude that not patching vulnerabilities has a significant negative effect on the honeymoon period. Of course there is no way to measure exactly when an attacker is likely to test an existing exploit against a newly released product however, the Sophoslabs [35] tests are indicative of how quickly a vendor might expect attackers to act.

5. RELATED WORK

As noted in the Introduction, both the scale of modern software systems and the scale of their deployment have made software design and engineering the focus of significant attention from scientists and engineers.

Brook's "The Mythical Man-Month" [5] is a bedrock reference for both the problems that the software engineering discipline is intended to address and its collected data (albeit from the 1960s) in support of its cogent observations. As Brooks addresses the issues in successfully engineering large software systems his focus is software defects ("bugs") rather than software security vulnerabilities. His analyses of the management issues in software engineering, particularly factors to account for in scheduling, still hold true. For example, the discussion of "Regenerative Schedule Disaster" (particularly Fig. 2.8, illustrating the added cost for training time) lends support to our observations about the time required to gain familiarity with a software system. Brook's Figure 11.2, "Bug occurrence as a function of release age", reproduced here on the left of Figure 1, shows an interval of decrease in bugs found, slowing to some minimum rate, followed by a slow rise in the rate of bugs found. This shows the effects of increased familiarity with a software system. As do many software engineering scholars, Brooks emphasizes the positive aspects of reusable software components without discussion of the potential risks from malicious actors.

Software reliability analysis is crucial to commercial firms which must deliver reliable software in a timely manner. A number of software reliability [21, 12, 27, 22] models have been developed, with a focus on bug rates and their implications for software maturity and releasability. The models, testing [26] and data collections do not address malicious actors.

Arbaugh, *et al.* [2] initiated the study of the more specialized software vulnerability life-cycle, with a particular focus on the intervals of time between when a vulnerability is known and when a software system is updated to remove the vulnerability. It is important to note, that these works focused on rate of exploitation, while this paper focuses on rate of vulnerability discovery.

Work by Jonsson, *et al.* [17] provides observations on a user population of students with quantitative evaluation of behavioral hypotheses, of which the most interesting to us is the ability to find bugs rapidly once the price is paid (in time) of learning the software system.

Alhamzi, et al [1] studied Windows 98 and Windows NT 4.0 and proposed a 3-phase S-shaped model (AIM) to describe the rate of change of cumulative vulnerabilities over time where the first phase includes time spent learning, but Ozment's analysis [24] of this and other vulnerability discovery models showed that its predictive accuracy assumed a static code-base and therefore was never tested against software spanning multiple versions. Our analysis supports an S-shaped curve model, but shows that the three phases in the AIM model do not accurately describe the data we have collected. Additionally, we are not concerned with the total number of vulnerabilities found over a product's lifetime, but with the first vulnerability found per version, as well as with a comparison of the cumulative number of days between vulnerabilities, particularly those closest to the product's release date.

Recent studies of bugs or vulnerabilities in large open source software systems [6, 25] did analyze vulnerability density across several versions and provide some data and observations that we believe support our hypothesis. First, since the software systems under study are open source software (e.g., Linux and OpenBSD) and readily available, they are learn-able by an attacker with an appropriate expenditure of time. Second, an analysis of bugs that persisted from version to version showed that such bugs were often a consequence of "cut and paste" software engineering, a crude yet effective form of software reuse. The majority of the existing vulnerability life-cycle and VDM research which makes use of the NVD dataset focused primarily on a small number of operating systems or a few server applications and in all but a few cases [25] only looked at one particular version of each (e.g. Windows NT, Solaris 2.5.1, FreeBSD 4.0 and Redhat 6.2, or IIS and Apache). In particular, Ozment and Schecter [25] found that 62% of the vulnerabilities in OpenBSD v.2.3-3.7 came from legacy code, and concluded that the original version of the source code may constitute the bulk of the later version's code base.

One large scale attempt to positively alter the rate of vulnerability discovery early on is Microsoft's Security Devel-

opment Lifecycle (SDL) which claims to have reduced the numbers of vulnerabilities found in Windows Vista's first year compared with those found in Windows XP, which does not use the SDL, (66 vs. 119) a 45% improvement. However, while Vista was in its first year, XP had been out for 6 years. We believe this also supports our hypothesis, especially since, in its first year, XP had only 28 vulnerabilities [20], a difference of 58%. [23]

Code reuse continues to be considered an important part of secure, efficient software development in both open and closed products [13, 10, 4]. However, Coverity's analysis of the lessons learned after years of using their static code analysis tool provides some possible explanations of the role legacy code plays in the honeymoon effect. For example, the authors list the most common response from software developers after the discovery of 1000+ bugs: "...The baseline is to record the current bugs, don't fix them, but do fix any new bugs... A reasonable conservative heuristic is if you haven't touched the code in years, don't modify it (even for a bug fix) to avoid causing any breakage." [3] This suggests that an attacker familiar with the legacy code that has been carried over into a newly released version would have an edge in finding new vulnerabilities in it (the legacy code), and this might have a negative effect on the honeymoon period.

In a recently published paper [28] the author analyzed the risk of first exploitation attempt using a Cox proportional model and concludes "that the exploitation process is accelerated for open source products". The focus of the paper is on measuring the rate of exploitation attempts, not on the rate of vulnerability discovery and is therefore not relevant to our paper.

6. DISCUSSION AND CONCLUSIONS

The software lifecycle has been repeatedly examined, with the intent of understanding the dynamics of software production processes, most particularly the arrival rate of software faults and failures. These rates decrease with time as updates gradually repair the errors as they are found, until an acceptable error rate is achieved.

The software vulnerability lifecycle has been less extensively studied, with most attention paid to the period after an exploit has been discovered. In attempting to understand the properties of vulnerability discovery, there are two approaches we might have taken. One approach would have been to study a single software system in depth, over an extended period, draw detailed conclusions, and perhaps generalize from them. Indeed, several of the related works mentioned above try to do just that for the middle and end phases of the lifecycle. But, another approach is to examine a large set of software systems and try to find properties that are true over the entire set and over an extended period.

We chose the latter approach for an number of reasons, which include the following: This approach allowed us to incorporate both open and closed source systems in our analysis, this approach also allowed us to analyze several different classes of software (Operating Systems, Web Browsers User applications, Server applications, etc), and this approach allowed us to discover general vulnerability properties, e.g. the honeymoon period, independent of the type of software, and without requiring a detailed analysis of the properties of each specific, individual vulnerability.

It might appear that given so many changes in tools, utilities, methodologies and goals used by both attackers and defenders over the last decade, a long term analysis would be inconsistent. To mitigate this we broke down each analysis by year and from version-to-version which are much shorter time intervals, and we demonstrated the consistency of this approach over time.

We also analyzed the role of legacy code in vulnerability discovery and found surprisingly, based on a detailed study of a large database of software vulnerabilities, that software reuse may be a significant source of new vulnerabilities. We determined that the standard practice of reusing code offers unexpected security challenges. The very fact that this software is mature means that there has been ample opportunity to study it in sufficient detail to turn vulnerabilities into exploits.

There are multiple potential causal mechanisms that might explain the existence of the honeymoon effect and the role played by familiarity. One possibility is that a second vulnerability might be of similar type to the first, so that finding it is fascilitated by knowledge derived from finding the first one. A second possibility is that the methodology or tools developed to find the first vulnerability lowers the effort required to find a subsequent ones. A third possible cause might be that a discovered vulnerability would signal weakness to other attackers (ie, blood in the water), causing them to focus more attention on that area. [7]

The first two possible causes require familiarity with the system, while the third is an example of properties *extrinsic* to the quality of the source code that might affect the length of the honeymoon period. An examination of these possible causes will appear in future work.

The period between when the error rate is low enough for release and attacker familiarity becomes high enough for an initial 0-day vulnerability we have called the *honeymoon* and its dynamics have been demonstrated in this paper to apply to the majority of popular software systems for which we had data.

The dynamics of the honeymoon effect suggest an interesting tradeoff between decreasing error rate and increasing familiarity with the software by attackers. This basic result has important implications for the arms race between defenders and attackers.

First, it suggests that a new release of a software system can enjoy a substantial *honeymoon* period without discovered vulnerabilities once it is stable, *independent of security practices*. Second, this honeymoon period appears to be a strong predictor of the approximate upper bound of the vulnerability arrival rate. Third, it suggests (as hinted at by the paper title) that attacker familiarity is a key element of the software process dynamics, and this is a contraindication for software reuse, as the greater the fraction of software reuse, the smaller the amount of study required by an attacker. Fourth, it suggests the need for more alternative approaches to security software systems than simply trying to create bug-free code.

In particular, research into alternative architectures or execution models which focuses on properties extrinsic to software, such as automated diversity, redundant execution, software design diversity [8] might be used to extend the honeymoon period of newly released software, or even give old software a *second honeymoon*.

6.1 Acknowledgments

Professors Blaze and Smith's work was supported by the

Office of Naval Research under N00014-07-1-907, Foundational and Systems Support for Quantitative Trust Management; Professor Smith received additional support from the Office of Naval Research under the Networks Opposing Botnets effort N00014-09-1-0770, and from the National Science Foundation under CCD-0810947, Blue Chip: Security Defenses for Misbehaving Hardware. Professor Blaze received additional support from the National Science Foundation under CNS-0905434 TC: Medium: Collaborative: Security Services in Open Telecommunications

References

[1] O.H. Alhamzi and Y.K. Malaiya. Modeling the vulnerability discovery process. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering(ISSRE'05)*, Washington, DC, USA, 2005.

[2] William A. Arbaugh, William L. Fithen, and John McHugh. Windows of vulnerability: A case study analysis. *Computer*, 33(12):52–59, 2000.

[3] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[4] BlackDuck. Koders.com. `http://corp.koders.com/about/`, April 2010.

[5] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley Professional, August 1995.

[6] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings, 18th ACM Symposium on Operating Systems Principles*, pages 73–82, October 2001.

[7] Sandy Clark, Matt Blaze, and Jonathan Smith. Blood in the water: Are there honeymoon effects outside software? In *In Proceedings of the 18th Cambridge International Security Protocols Workshop -pending publication*. Springer, 2010.

[8] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *In Proceedings of the 15th USENIX Security Symposium*, pages 105–120, 2006.

[9] CVE. Common vulnerabilities and exposures, 2008.

[10] Dr Dobbs Journal. Open Source Study Reveals High Level of Code Reuse. `http://www.drdobbs.com/open-source/216401796`, March 2009.

[11] Stefan Frei. *Security Econometrics - The Dynamics of (In)Security*. Eth zurich, dissertation 18197, ETH Zurich, 2009. ISBN 1-4392-5409-5, ISBN-13: 9781439254097.

[12] A.L. Goel and K. Okumoto. A time dependent error detection model for software reliability and other performance measures. *IEEE Transactions on Reliability*, R-28:206–211, August 1979.

[13] Michael Howard and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, May 2006.

[14] IBM Internet Security Systems - X-Force. X-Force Advisory. http://www.iss.net.

[15] iDefense. Vulnerability Contributor Program. http://labs.idefense.com/vcp.

[16] Pankaj Jalote, Brendan Murphy, and Vibhu Saujanya Sharma. Post-release reliability growth in software products. *ACM Trans. Softw. Eng. Methodol.*, 17(4):1–20, 2008.

[17] Erland Jonsson and Tomas Olovsson. A quantitative model of the security intrusion process based on attacker behavior. *IEEE Trans. Softw. Eng.*, 23(4):235–245, 1997.

[18] M.C. McIlroy. Mass producted software components. *Report to Scientific Affairs Division, NATO*, October 1968.

[19] Microsoft. Internet explorer architecture. `http://msdn.microsoft.com/en-us/library/aa741312(VS.85).aspx`, 2010.

[20] Microsoft Corporation. Microsoft security development lifecycle. `http://www.microsoft.com/security/sdl/benefits/measurable.aspx`, September 2008.

[21] John D. Musa. A theory of software reliability and its application. *IEEE Transactions on Security Engineering*, SE-1:312–327, September 1975.

[22] John D. Musa, Anthony Iannino, and Kasuhira Okumoto. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill, 1987.

[23] NIST. National Vulnerability Database, 2008.

[24] Andy Ozment. Improving vulnerability discovery models. In *QoP '07: Proceedings of the 2007 ACM workshop on Quality of protection*, pages 6–11, New York, NY, USA, 2007. ACM.

[25] Andy Ozment and Stuart E. Schechter. Milk or wine: does software security improve with age? In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

[26] R.E. Prather. Theory of program testing - an overview. *Bell System Technical Journal*, 72(10):3073–3105, December 1983.

[27] C.V. Ramamoorthy and F.B. Bastani. Software reliability - status and perspectives. *IEEE Transactions on Software Engineering*, SE-8(4):354–371, July 1982.

[28] Sam Ransbotham. An Empirical Analysis of Exploitation Attempts based on Vulnerabilities in Open Source Software. In *Workshop on the Economics of Information Security (WEIS)*, June 2010.

[29] Secunia. http://www.secunia.com. Vulnerability Intelligence Provider.

[30] Security Focus. Vulnerabilities Database, 2008.

[31] SecurityTracker. `http://www.SecurityTracker.com`. SecurityTracker.

[32] TippingPoint. Zero day initiative (zdi). `http://www.zerodayinitiative.com/`.

[33] US-CERT. Vulnerability statistics. `http://www.cert.org/stats/vulnerability\_remediation.html`.

[34] Vupen. Vupen security. `http://www.vupen.com`.

[35] Chester Wisniewski. Windows 7 vulnerable to 8 out of 10 viruses, 2009. `http://www.sophos.com/blogs/chetw/g/2009/11/03/windows-7-vulnerable-8-10-viruses/`.

Quantifying Information Leaks in Software

Jonathan Heusser
School of Electronic Engineering and Computer Science
Queen Mary University of London
jonathan.heusser@dcs.qmul.ac.uk

Pasquale Malacaria
School of Electronic Engineering and Computer Science
Queen Mary University of London
pm@dcs.qmul.ac.uk

ABSTRACT

Leakage of confidential information represents a serious security risk. Despite a number of novel, theoretical advances, it has been unclear if and how quantitative approaches to measuring leakage of confidential information could be applied to substantial, real-world programs. This is mostly due to the high complexity of computing precise leakage quantities. In this paper, we introduce a technique which makes it possible to decide if a program conforms to a quantitative policy which scales to large state-spaces with the help of bounded model checking.

Our technique is applied to a number of officially reported information leak vulnerabilities in the Linux Kernel. Additionally, we also analysed authentication routines in the Secure Remote Password suite and of a Internet Message Support Protocol implementation. Our technique shows when there is unacceptable leakage; the same technique is also used to verify, for the first time, that the applied software patches indeed plug the information leaks.

This is the first demonstration of quantitative information flow addressing security concerns of real-world industrial programs.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Information flow controls; D.2.4 [**Software/Program Verification**]: Model checking, Correctness proofs; H1.1 [**Systems and Information Theory**]: Information theory

General Terms

Security, Theory

Keywords

Information leakage, Linux kernel, Quantitative information flow

1. INTRODUCTION

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

Quantitative Information Flow (QIF) [3, 11] aims to provide techniques and tools able to quantify leakage of confidential information. As a motivating example consider a prototypical password checking program

```
if (password==guess) access=1 else access=0
```

Notice how there is an unavoidable leakage of confidential information in this program: an attacker observing the value of `access` will be able to infer if he guessed the right password (complete leakage if he did guess it right) and if the guess was wrong he will have eliminated one possibility from the search space. Notice also how essential the *amount of information leaked* is: if the amount leaked is very small then the program could as well be considered secure.

If, as the above example illustrates, leakage is somehow unavoidable then the real question is not whether or not programs leak, but how much. This point is what makes Quantitative Information Flow an appealing theory. In a nutshell, QIF aims to measure the amount of information from confidential data (in the above example the variable `password`) that an attacker who can read/write the public input data (`guess`) will be able to infer from some observable variable (`access`).

However, implementing a precise QIF analysis for secret sizes of more than a few bits is computationally infeasible; roughly speaking this is because classical QIF computes the entropy of a random variable whose complexity is the same as computing all possible runs of the program. Even when abstraction techniques and statistical sampling are integrated with QIF [9] to help the scalability issues a useful analysis for real code still seems problematic.

In this paper, we introduce a useful quantitative analysis for C code: we will demonstrate the analysis on reported information leakage vulnerabilities in the Linux Kernel and in common authentication routines. All of the covered vulnerabilities are referenced by the standardised vulnerability repository CVE from Mitre[1].

To address the computational feasibility of the quantitative analysis we shift the focus from the question "How much does it leak?" to the simpler quantitative question "Does it leak more than k?". We will show how the questions are related and more importantly we will show that off-the-shelf symbolic model checkers like CBMC [5] are able to efficiently answer the second kind of question. CBMC is a good choice for several reasons: (i) it makes it easy to parse and analyse

[1]http://cve.mitre.org, CVE is industry-endorsed with over 70 companies actively involved.

large ANSI-C based projects (ii) it models bit-vector semantics of C accurately which makes it able to detect arithmetic overflows amongst others, which turns out to be important (iii) nondeterministic choice functions can be used to easily model user input, which also enjoys efficient solving due to the symbolic nature of the model checker (iv) despite being a bounded model checker, CBMC can check whether enough unwinding of the transition system was performed to prove that there are no deeper counterexamples.

Our experiments show that the analysis not only quantifies the leakage but also helps in understanding the nature of the leak. In particular, the counterexample produced by the model checker, when a leakage property is violated, can provide insights into the cause of the leak. For example, we can extract a public user input from the counterexample needed to trigger a violation.

Another surprising result of our experiment is that in certain circumstances we were able to use our technique to prove whether the official patch provided for the vulnerability does actually eliminate the information leak. This is achieved by point (iv) from above, when the model checking process is actually complete.

In summary the main technical contributions of this paper are the following:

1. We present the first quantitative leakage analysis of systems software.

2. We show how to express Quantitative Information Flow properties that can be efficiently checked using bounded symbolic model checking.

3. We show that the technique not only quantifies leakage in real code but also provides valuable information about the nature of the leak.

4. In some cases we are able to prove that official patches for reported vulnerability do indeed eliminate leakage; these constitute the first positive proofs of absence of QIF vulnerabilities for real-world systems programs.

2. MODEL OF PROGRAMS AND DISTINCTIONS

We aim to model the input/output behaviour of a C function where inputs are formal arguments to the function and outputs are either return values or pointer arguments.

In the following we will consider P to be a C function taking high and low inputs noted h, l; we call observables low variables whose values are "publicly available" after running P. As an example consider the following "modulo" program

```
o = (h % 4 ) + l
```

and suppose `h` is a 4 bits variable with values 0..15 and `l` a 1 bit variable with values 0,1; then the low input for P is the variable l and the observable is the variable o whose possible values are $0 \dots 4$.

Formally, a program P is modelled as transition system $TS = (S, T, I, F)$ with S being the program states, $T \subseteq S \times S$ are the program transitions and I the initial states and F the final states. Let us define a successor function for a state $s \in S$

$$\text{Post}(s) = \{s' \in S \mid (s, s') \in T\}$$

A state s is in F if $\text{Post}(s) = \emptyset$. A path is a finite sequence of states $\pi = s_0 s_1 s_2 \dots s_n$ such that $s_0 \in I$ and $s_n \in F$.

A state is a tuple $S = S_H \times S_L$ of the pair of confidential input H and low input L. We consider initial/final or input/output pairs of states of a path, $\langle (h, l), o \rangle$ where the second component is the output o produced by the final state drawn from some output alphabet O. In the above example an input/output pair would be $\langle (5, 1), 2 \rangle$ representing the computation $(5\%4) + 1 = 2$.

Confidential inputs are denoted as $h \in H$, low inputs $l \in L$, and low observations $o \in O$, where the output behaviour of the function is always a low observation and the input is an initial state (h, l). A distinction on the confidential input through observations O is one where there exists at least two paths through P, modelled as TS, which leads to different observations for different confidential input but constant low input.

We define an equivalence relation $\simeq_{P,l}$ on the values of the high variables as follows: $h \simeq_{P,l} h'$ iff if $\langle (h, l), o \rangle$, $\langle (h', l), o' \rangle$ are input/output pairs in P then $o = o'$.

Hence, two high values are equivalent (w.r.t. a low value l) if they cannot be distinguished by any observable. In the running example an equivalence class in $\simeq_{P,1}$ would for example be $\{1, 5, 9, 13\}$. The equivalence relation associated to P, l is an element of the set of all possible equivalence relation on the values of high.

Let $\mathcal{I}(X)$ be the set of all possible equivalence relations on a set X. Define on $\mathcal{I}(X)$ the order:

$$\approx \sqsubseteq \sim \leftrightarrow \ \forall s_1, s_2 \ (s_1 \sim s_2 \Rightarrow s_1 \approx s_2) \qquad (1)$$

where $\approx, \sim \in \mathcal{I}(X)$ and $s_1, s_2 \in X$. $\sqsubseteq$ defines a complete lattice over X. It is a *refinement order* with bottom element being the relation relating every state and top element being the identity relation. This is described as the Lattice of Information [10].

Non leaking programs (i.e. satisfying non-interference [7]) are characterised as follows:

PROPOSITION 1. *P is non-interfering iff for all l, $\simeq_{P,l}$ is the least element in $\mathcal{I}(S_H)$.*

An attacker controlling the low inputs can be modelled by an equivalence relation $\simeq_P$ corresponding to a particular $\simeq_{P,l}$.

Formally, we define a *quantitative policy* as a non-negative natural number N. A relation $\simeq_{P,l}$ breaches a policy if $\mid \simeq_{P,l} \mid > N$ (where $\mid \simeq_{P,l} \mid$ is the number of equivalence classes of $\simeq_{P,l}$). In our model, an attacker will always choose a relation breaching the policy, provided that given a policy and a program such a relation exists. We use $\simeq_P$ with the program P being initialised with the attacker's choice of l[2].

In the above example, a choice could be $\simeq_P = \simeq_{P,0}$ corresponding to the program `l=0; o = (h % 4 ) + l`.

Quantitative Information Flow uses information theoretical measures like Shannon entropy to measure leakage of confidential information. The measure of a program can be broken down into two main steps [11, 8]:

1. interpret the program as a random variable R_P

2. compute the entropy of R_P (noted $H(R_P)$)

[2]In the paper such attacker choices will be modelled by the nondeterministic choice function `input()`.

It has been shown that R_P and $\simeq_P$ coincide [11, 13]. For example for the modulo program above under the assumption of uniform distribution on the input there are 4 equivalence classes each having probability $\frac{1}{4}$. The Shannon entropy of that program is then

$$4 * -\frac{1}{4}\log_2(\frac{1}{4}) = 2$$

This number 2 represents the fact that the observations reveal which of the 4 possible classes (i.e. 2 bits of information) the high input belongs to.

R_P and $\simeq_P$ are also order related as the following proposition shows [8]:

PROPOSITION 2. *$\simeq_P \sqsubseteq \simeq_{P'}$ iff for all probability distributions $H(R_P) \leq H(R_{P'})$*

To further understand the importance of $\simeq_P$ in Quantitative Information Flow we need to introduce the information theoretical concept of channel capacity: consider the password check example from the introduction. Suppose the password is a 64 bits randomly chosen string; we have two equivalence classes, one with 1 element so having probability $\frac{1}{2^{64}}$, the other class with $2^{64}-1$ elements having thus probability $1-\frac{1}{2^{64}}$. The entropy is then 3.46944695×10^{-18}: as expected a password check of a big password should leak very little. Suppose however that the probabilities of the high inputs are such that both equivalence classes have probability $\frac{1}{2}$. Then the entropy dramatically raises to 1 which is the channel capacity, i.e. the maximum leakage achievable given two classes: $\log_2(2) = \log_2(|\simeq_P|)$. In the modulo example the channel capacity is 2 which happens to be given by the uniform distribution on the high input. Other distributions on the high input cannot give higher entropy: for example if we consider the distribution where all even numbers have equal probability $\frac{1}{8}$, and all odd numbers have 0 probability then the resulting entropy will be 1.

The following result establishes basic relationships between leakage, channel capacity, and number of distinctions:

PROPOSITION 3.

1. *P is non-interfering iff $\log_2(|\simeq_P|) = 0$*
2. *The channel capacity[3] of P is $\log_2(|\simeq_P|)$.*
3. *If for all probability distributions $H(R_P) \leq H(R_{P'})$ then $|\simeq_P| \leq |\simeq_{P'}|$*

Point (1) is proved in [4], (2) in [12] and (3) is a consequence of proposition 2 whose proof is in [8]. Hence a lower bound on $|\simeq_P|$ provides a lower bound on the channel capacity of the program P.

Hence, because of proposition 3 the inequality $|\simeq_P| > N$, which is at the heart of our analysis, can be rephrased to the following statement: in a setting where the distribution of the secret is the most favourable for the attacker then the leakage is at least $\log_2(N)$ bits.

3. ENCODING DISTINCTION-BASED POLICIES

[3]The channel capacity is the maximum possible leakage where we consider all possible probability distributions on the inputs [12]

Recall that for a program P a quantitative policy is a natural number N which limits the cardinality of $\simeq_P$ to N.

In other words, a program violates a quantitative policy if it makes more distinctions than what is allowed in the policy. A leaking program is one breaching the policy $N = 1$ in the above definition.

We take ideas from assume-guarantee reasoning [17] to encode such a policy in a driver function, which tries to trigger a violation, i.e. producing a counterexample, of the policy. If the policy states that the function `func` is not allowed to make more than 2 distinctions then this is modelled as shown in Program 1. This driver only has a high component as a state, which is passed to the function `func` where the policy is tested on.

```
int h1,h2,h3;
int o1,o2,o3;

h1 = input();  h2 = input(); h3 = input();

o1 = func(h1);
o2 = func(h2);
assume(o1 != o2); // (A)

o3 = func(h3);
assert(o3 == o1 || o3 == o2); // (B)
```

Program 1: Example driver checking for 2 distinctions

Drivers always have a similar structure: we model the secret by a nondeterministic choice function `input()` as a placeholder for all possible values of that type; then for a policy of checking for N distinctions, the function under inspection is called N times. The crucial step (A) is the use of the `assume` statement after the calls: the driver assumes that, in this case, there are two different return values found already. The function is called an $N+1$th time and at (B) the driver asserts that the next output is either one of the previously found outputs.

The `assume` statement only considers execution paths which satisfy the given boolean formula, all other paths are rejected. Further, the bounded model checker used will try to find a counterexample to the negated assertion claim, which is only satisfiable if and only if a counterexample exists. An unsatisfiable formula means that the original claim holds, i.e. the program conforms to the policy. The verification condition generated by the bounded model checker for the policy in Program 1 is:

$$\texttt{o1 != o2} \implies \texttt{(o3 == o1 || o3 == o2)}$$

Where the bounded model checker tries to find a counterexample (execution path) using the negated claim such that the following holds

$$\texttt{o1 != o2} \wedge \texttt{o3 != o1} \wedge \texttt{o3 != o2}$$

i.e. that there are three distinctions possible.

Another possibility is that the function `func` does not even make two distinctions, such that the `assume` statement at point (A) is always false, which leads to proving the policy (or any policy) vacuously true, because for any assertion `Q` the verification condition is true, i.e. $\texttt{false} \implies \texttt{Q}$.

3.1 Bounded Model Checking

We use the bounded model checker CBMC to verify or falsify a policy. CBMC encodes an ANSI-C program into a

Input: Function `func`, types `t,t',t"`, comparison `eq_t`, bound k, threshold N
Output: Driver.c

```
t o_1, ..., o_n, o_n+1;
t' h_1, ..., h_n, h_n+1;
t'' l;

h_1 = input(); ... h_n = input();
l = input();
o_1 = func(h_1, l);
⋮
o_n = func(h_n, l);
assume(!eq_t(o_1, o_2) && !eq_t(o_1, o_3) && ...);

o_n+1 = func(h_n+1, l);
assert(eq_t(o_n+1, o_1) || eq_t(o_n+1, o_2) || ...);
```

Algorithm 1: Template to syntactically generate a driver for an N distinction policy

propositional formula by unwinding the transition relation and user defined specifications up to some bound. This formula is only satisfiable if there exists an error trace violating the specification.

The tool can also check if the unwinding bound is sufficient by introducing *unwinding assertions*, which are assertions on the negated loop guards. This ensures that no longer counterexample can exist than the used bound. To *prove* any properties the analysis has to pass unwinding assertions, otherwise it can only be used as a way to find counterexamples up to the unwinding bound.

The C program gets encoded into constraints C and the property – user defined assertions – are encoded in P. Then the model checker tries to find a satisfiable assignment to the formula

$$C \wedge \neg P$$

where P is an accumulation of the assumptions and assertions made in the program text. Thus if there are two `assume` statements in the driver with expressions E_1 and E_2 and one `assert` statement with expression Q then P is

$$P \equiv E_1 \wedge E_2 \implies Q$$

3.2 Driver

A general template for a driver is described in Algorithm 1. The inputs to the algorithm are the function `func` to be analysed, possibly up to three different types for the input/output pair $\langle (h, l), o \rangle$, and a comparison function `eq_t` which returns true if the arguments of type `t` are equal, where `t` is the type of the observation of function `func`. This comparison function could be as simple as `==` of C, or a more complex function, such as `memcmp`, if `t` is an array or string. Also note, that the observations `o_i` do not need to be only return values, but can also be pointer arguments to `func`.

PROPOSITION 4 (CORRECTNESS OF DRIVER TEMPLATE). *If the driver template in Algorithm 1 is successfully verified up to a bound k (i.e. the negated claim is unsatisfiable) then the function `func` does not make more than N distinctions on the output within the bound k. Formally, we state that the validity of the driver implies the validity of the following implication*

$$o_1 \neq o_2 \wedge o_1 \neq o_3 \wedge \cdots \wedge o_{n-1} \neq o_n \implies o_{n+1} = o_1 \vee \cdots \vee o_{n+1} = o_n$$

Thus, we can make the following claims on the result of the model checking process: For a given bound k and a policy,

- if the model checker finds a counterexample then the policy is violated, i.e. the program makes more distinctions than specified
- if the process ends with a successful verification of the policy without unwinding assertions then the policy holds up to an unwinding of k.
- if the process ends with a successful verification of the policy *with* unwinding assertions then the policy holds for any number of iterations.

4. CHECKING QUANTITATIVE POLICIES

The steps in checking a program or function for the compliance with a quantitative policy are as follows: (1) Define the input state (h, l) and output state o in the code, i.e. the confidential input h, the low input l and the observation o (2) Define the maximum number of distinctions in the policy and an unwinding factor k (3) Generate a driver function using the template in Algorithm 1 (4) Run CBMC on the driver. If the driver is successfully verified, potentially increase the unwinding factor.

4.1 Modelling Low Input

A crucial aspect of our analysis is to model low user input, which is most of the time responsible for triggering a bug which causes the information leak. These bugs only happen on a very restricted number of execution paths and could be exploited by a malicious user choosing a special user input. This scenario generally applies when studying many CVE reported information leakage vulnerabilities.

Let us look at the following simplified code in Program 2, which contains an integer underflow, taken from the vulnerability CVE-2007-2875 in the linux kernel.

```
typedef long long loff_t;
typedef unsigned int size_t;
int underflow(int h, loff_t ppos) {
  int bufsz;
  size_t nbytes;
  bufsz=1024;
  nbytes=20;

  if (ppos + nbytes > bufsz) // (A)
       nbytes = bufsz - ppos; // (B)
  if(ppos + nbytes > bufsz) {
     return h; // (C)
  } else {
     return 0;
  }
}
```

Program 2: Integer underflow causing a leak

At first, it seems not possible that the point (C) where the secret `h` gets returned is ever executed, because exactly that check is done in (A) which reduces the variable `nbytes` to

be within the bound `bufsz`. However, due to wrong choice and combination of types, the subtraction in (B) causes an underflow in `nbytes` for a very large `ppos` value. And unfortunately, `ppos` is a user controlled input variable, such that when its value is chosen correctly, point (C) is reached.

In this case, a state in the system is the tuple (h, l) which represents the arguments to the function `underflow`, i.e. the formal parameters `h` and `ppos`; observations are the return values of this function. The generated driver can automatically find the low part of a state which triggers such subsequent information leaks, because the analysis instructs the model checker to find *any* possible execution path satisfying the assumptions and assertions on the outputs, given nondeterministic high values and fixed low inputs. As SAT-based model checking is precise down to the individual bit, it will find a low input which triggers the underflow and uncovers the leak.

CBMC generates a counterexample falsifying a policy of e.g. no leakage and thereby having triggered the integer underflow. The following excerpt of the counterexample

```
State 14 file underflow.c line 40 function main
------------------------------------------------------
  underflow::main::1::l=1706688912 (00000000...
....
State 35 file underflow.c line 13 function underflow
------------------------------------------------------
  underflow::underflow::1::nbytes=4027596816 (11110000...
```

shows that a low input of `l=1706688912` lead to an `nbytes` which underflowed from the previous value 20.

Clearly, for such leaks to be detected it needs bit-level precise reasoning, just like SAT-based bounded model checkers support.

4.2 Environment

In model checking, the environment, like library function calls or generally functions and data structures which have no implementation, need to be modelled in a way which allows for the property to be verified. Out of the box, CBMC replaces function calls with no implementation with nondeterministic values.

As our analysis needs to check for equality on inputs and outputs of functions a certain number of common library functions have to be modelled in a way which preserves their original semantics. For example, the usual library C functions `memcmp`, and `strcmp` are implemented in a way which return 0 if their arguments are equal and a value not equal to 0 if they are not equal. The functions `memset` and `memcpy` actually set an array of integers or characters to a certain value or to the content of another array. The same applies to linux kernel utility functions such as `copy_to_user` and `copy_from_user` which copy memory blocks to or from userspace.

For example, a `memcmp` implementation is shown in Program 3.

```
int memcmp(char *s1, char *s2, unsigned int n) {
  int i;
  for(i=0;i<n;i++) {
     if(s1[i] != s2[i]) return -1;
  }
  return 0;
}
```

Program 3: Simplified `memcmp` model

5. EXPERIMENTAL RESULTS

We applied our technique to CVE reported information leakage vulnerabilities in the Linux Kernel. In the experiments we checked for policy violations and proved whether official patches resolve the information leakage. We also analysed authentication routines of the Secure Remote Password protocol (SRP) and of a Internet Message Support Protocol implementation. A summary of the results is shown in Table 1. The leakage is reported in the second last column where $> \log_2(N)$ means that more than $\log_2(N)$ bits leaked, i.e. the policy N has been violated; equally, $\leq \log_2(N)$ means the policy N has been verified. These two cases correspond to lower and upper bounds on the leakage.

5.1 Linux Kernel

We define information leakage in the kernel always as parts of the kernel memory which gets mistakenly copied to user space, i.e. the virtual memory allocated to conventional applications. Clearly, this should not happen as anything allocated in the kernel space is not meant to be seen by users (except within the bounds of normal user/kernel interactions), especially in multi-user systems like Linux. Thus, in all examples the kernel memory is modelled as nondeterministic values.

The interface between user and kernel space are system calls or syscalls in short. Syscalls, like normal functions, have a number of arguments and a return value where the kernel can transfer data structures or single values back and forth. This is the crucial point in the system where information leakage is most common.

AppleTalk. The specific vulnerability CVE-2009-3002 in the appletalk network code shows a quite common cause of information leakage: a user requests, by a syscall, that a structure gets filled with values and returned to user land. The developer however forgot to assign values to all fields in the struct, thus these missing fields get "filled" with unspecified kernel memory, as it is allocated on the stack. This CVE security bulletin actually comprises six different vulnerable network protocol implementations, all following the same leakage pattern. We will only present the affected code of the AppleTalk implementation – the same kind of analysis applies to all six vulnerabilities.

In this case the structure returned to the user is shown in Program 4. The leaking function is `atalk_getname` in

```
struct sockaddr_at {
    u_char sat_len, sat_family, sat_port;
    struct at_addr      sat_addr;
    union {
        struct netrange r_netrange;
        char            r_zero[8];
    } sat_range;
};
#define sat_zero sat_range.r_zero
```

Program 4: Complex observation struct leads to leak from `sat_zero`.

`net/appletalk/ddp.c` is shown in Program 5.

In the function, the structure `sat` gets filled with values provided by the kernel, at the end the whole structure is copied via `memcpy` to the address of the `uaddr` pointer, which is indirectly, via the syscall `getsockname` copied back to user

Description	CVE Bulletin	LOC	$k^\star$	Patch Proof	$log_2(N)$	Time
AppleTalk	CVE-2009-3002	237	64	✓	>6 bit	1h39m
tcf_fill_node	CVE-2009-3612	146	64	✓	>6 bit	3m34s
sigaltstack	CVE-2009-2847	199	128	✓	>7 bit	49m50s
cpuset†	CVE-2007-2875	63	64	×	>6 bit	1m32s
SRP getpass	–	93	8	✓	≤1 bit	0.128s
login_unix	–	128	8	–	≤2 bit	8.364s

Table 1: Experimental Results. ⋆ Number of unwindings † From Section 4.1

```
int atalk_getname(struct socket *sock,
      struct sockaddr *uaddr, int *uaddr_len, int peer) {
        struct sockaddr_at sat;

        // Official Patch. Comment out to trigger leak
        //memset(&sat.sat_zero, 0, sizeof(sat.sat_zero));
        .
        ⋮ // sat structure gets filled
        memcpy(uaddr, &sat, sizeof(sat));
        return 0;
}
```

Program 5: Function introducing the leak for CVE-2009-3002.

land. However, the field `sat.sat_zero` has not been initialised, thus a number of bytes of kernel memory get copied back to the user.

The secret is implicitly modelled by allocating the `sat` structure with nondeterministic values; observations are also of type `sockaddr_at`. The driver uses as parameter `eq_t` the library function `memcmp` to compare memories.

The model checker found a counterexample for a 6 bit policy within 1 hour and 39 minutes. Once the official patch was applied of setting the `sat` structure to 0 with `memset`, our driver successfully verified the policy in about the same time with unwinding assertions, thus it proved that the patch stops the leak.

tcf_fill_node. This information leak happens in the netlink subsystem of the kernel. The function `tcf_fill_node` prepares a `struct tcmsg` to be sent back to the user. However, the programmer made a typing mistake and filled a field `tcm__pad1` twice instead of the second time for `tcm__pad2`.

```
struct tcmsg *tcm;
...
nlh=NLMSG_NEW(skb, pid, seq, event, sizeof(*tcm), flags);
tcm=NLMSG_DATA(nlh);
tcm->tcm_family = AF_UNSPEC;
tcm->tcm__pad1 = 0;
tcm->tcm__pad1 = 0; // typo, should be tcm__pad2 instead.
```

Program 6: Function excerpt introducing the leak for CVE-2009-3612.

This leaks kernel memory from `tcm__pad2` back to userspace. Here, we again modelled kernel memory implicitly by the memory allocated for `tcm` through the function `NLMSG_DATA`, which initialised the fields of the struct with nondeterministic values. The observation is the filled out variable `tcm`, the low user input is a simple integer variable not mentioned here for brevity.

The official patch which was applied to fix the leak is simply changing the last line above to `tcm->tcm__pad2=0`. We were again able to prove that this patch successfully fixes the security hole and otherwise the program violates a leakage policy of 6 bits.

Without the patch, a counterexample is found within 3 minutes and 34 seconds; with the patch, the program is verified within about the same time.

sigaltstack. The leakage for this vulnerability is intricate and only manifests itself on 64-bit processors. On such a system, the struct `stack_t`, as shown in Program 7, will be padded to a multiple of 8 bytes because on 64-bit systems `void*` and `size_t` are both 8 bytes (instead of 4 bytes for 32-bit systems), while an integer type remains 4 bytes. Thus, the size of `stack_t` is padded to 24 bytes, while on a 32-bit system it remains unpadded at 12 bytes.

```
typedef struct sigaltstack {
        void __user *ss_sp;
        int ss_flags; // 4 bytes padding on 64-bit
        size_t ss_size;
} stack_t;
```

Program 7: Structure with padding depending on architecture.

The syscall `do_sigaltstack` in `kernel/signal.c` copies such a structure back to userland via the copy function `copy_to_user`, however it does not clear the padding bytes, thus those are leaked to the user on a 64-bit system. In the function visible in Program 8, the high input is the structure `oss` and the low output is the argument `uoss`.

```
int do_sigaltstack (const stack_t __user *uss,
                stack_t __user *uoss, unsigned long sp) {
     stack_t oss;
      ... // oss fields get filled
      if (copy_to_user(uoss, &oss, sizeof(oss)))
             goto out; ....
```

Program 8: Leakage through copying whole structures including padding.

CBMC supports modelling of 64-bit widths however that is not enough to automatically measure the padding bytes. This is because the `sizeof` operator in CBMC returns only the sum of all sizes without eventual bit alignments. This is solved in our approach by providing a model of the `copy_to_user` function, just like e.g. an implementation of `memcpy` is provided, which checks if the length parameter is aligned according to the architecture (4 bytes for 32 and 8 bytes for 64). If there are padding alignments then these will be chosen to be filled with nondeterministic integer values modulo the number of padding bytes.

In Program 8, this would translate to the following: `sizeof(oss)` counts 20 bytes as the size of the structure. However, this does not account for the padding bytes, and our `copy_to_user` model does the following calculation:

```
pad = ALIGN - (sizeof(oss) % ALIGN);
if(pad == ALIGN)
    padding = 0;
else
    padding = ((unsigned int) nondet_int()) %
                    (1 << (pad*8))
```

where `ALIGN` is chosen to be 4 or 8 depending on the architecture used. In a 64-bit system, this translates to $8-(20\%8) = 4$ bytes for `pad` which are represented by the `padding` variable.

With this setup, we were able to verify that on a 32-bit system the Program 8 does not leak anything, while on a 64-bit system this violates a policy of e.g. 7 bits. A counterexample was found within 49 minutes and 50 seconds. We were also able to prove that the patch applied removes the padding leak. The patch in this case was to not copy the whole struct but copying the three struct members separately through the function `__put_user`, where the padding does not come into play.

cpuset. The crucial part of this vulnerability has already been discussed in Section 4.1. Our analysis finds the right low input which triggers the integer underflow. The actual code however does not simply return the secret as shown in the section mentioned above, but it copies `nbytes` number of bytes from a buffer `ctr->buf` at offset `*ppos`. Because of

```
if (*ppos + nbytes > ctr->bufsz)
      nbytes = ctr->bufsz - *ppos;
if (copy_to_user(buf, ctr->buf + *ppos, nbytes))
     return -EFAULT;
```

the underflow, `nbytes` and `*ppos` access memory way out of the actual buffer and thus disclose kernel memory. However our analysis of this vulnerability requires at the moment too much manual intervention to model memory access outside of the allowed bound (i.e. `ctr->buf + *ppos`).

One elegant way of addressing this problem would be by modifying CBMC itself; CBMC could for example return nondeterministic values for such out-of-bound memory accesses which would implicitly model the access to confidential data.

5.2 Authentication Checks

We analysed parts of the authentication routines of the secure remote password suite (SRP) and the Unix passwd-authentication of Cyrus' Internet Message Support Protocol daemon (IMSPD).

SRP. To demonstrate that confidential variables and observations can be used flexibly, we checked that there is no leakage in the password request function in `libsrp/t_getpass.c`.

The confidential input is the password entered by the user when being prompted at the login; the observations are the *echos* of the terminal of typed characters. Whether the terminal echos the typed characters or not depends on which mode the console is in. The environment modelling the console and its modes had to be provided to check this program.

```
_TYPE( int ) t_getpass (char* buf, unsigned maxlen,
                                  const char* prompt) {
    DWORD  mode;

    GetConsoleMode( handle, &mode );
    SetConsoleMode( handle, mode & ~ENABLE_ECHO_INPUT );
    if(fputs(prompt, stdout) == EOF ||
        fgets(buf, maxlen, stdin) == NULL) {
        SetConsoleMode(handle,mode);
        return -1;
    } ....
```

Program 9: Side-effect of `mode` decides on echo output of `fgets`

In Program 9, the function `t_getpass` first gets the current mode of the console by the function `GetConsoleMode`; then it sets a new console mode by inverting the bit `ENABLE_ECHO_INPUT` in the mode through the function `SetConsoleMode` which clearly disables the echo of input read from standard input. The function `GetConsoleMode` is modelled by nondeterministically setting the mode to any integer value, the function `SetConsoleMode` sets a global mode variable to its second argument. The function `fgets`, which reads a number of bytes from `stdin`, is modelled to return its first argument `buf` completely if the mode is set to echo the input and return a constant value otherwise.

With this setup CBMC proves through our driver that starting from any initial `mode`, the program will always end up with $\log_2(|\simeq_P|) = 0$, i.e. that there is no leakage. We can also successfully check that if the line which disables the echo is removed then the policy is violated.

IMSPD. The function checked in this test is `login_plaintext` in `imsp/login_unix.c` as shown in Program 10.

```
int login_plaintext(char *user, char* pass,
                           char** reply) {
    ...
    struct passwd* pwd = getpwnam(user);
    if (!pwd) return 1;
    if (strcmp(pwd->pw_passwd,
               crypt(pass, pwd->pw_passwd)) != 0) {
        *reply = "wrong password";
        return 1;
    }
    return 0;
```

Program 10: Login function of IMSPD.

The program first tries to receive the stored password context of a user using the function `getpwnam`. If successful, it will compare the stored with the entered password using `strcmp`. If this fails it will set the string `reply` to "wrong password". If authentication is successful it returns 0.

Clearly, this function has three distinguishable observations: (1) it returns 1 (2) it returns 1 and sets `*reply` (3) it returns 0. We modelled the three parameters to the function as low user input and the stored password as confidential variable. With this setup, we are able to verify that this program conforms to a policy which only leaks 3 observations, within 9 seconds.

6. RELATED WORK

There have been several attempts in recent years to build a quantitative analysis of leakage, starting with the static

analysis in [4].

The most relevant works for this paper are [1] by M. Backes, B. Köpf and A. Rybalchenko and [8] by J. Heusser and P. Malacaria where verification techniques are used to compute leakage of programs. Those works are both inspired by the important previous theoretical work on self composition by G. Barthe, P. D'Argenio, and T. Rezk [2] and T. Terauchi and A. Aiken [18]. However as already noted, those approaches attempt primarily to answer questions about how much a program leak and seem unable to scale to real code in terms of line of code, state space and language constructs. In particular, they have not, as far as we are aware, been used to analyse independently existing vulnerabilities in independently existing programs.

On the theoretical side, the complexity of QIF analysis has recently been thoroughly investigated by H. Yasuoka and T. Terauchi [19] who, amongst other aspects, explored the relation to verification and k-safety properties.

Approaches that do scale to large programs are by S. McCamant, M. D. Ernst [14] and J. Newsome, S. McCamant, D. Song [15]. They released an impressive tool, FlowCheck, which is able to analyse very large programs. There are however significant differences between the approaches in that FlowCheck is a security testing tool based on the Valgrind dynamic instrumentation framework whereas our approach is based on verification and static analysis techniques. Thus, our work comes with stronger theoretical guarantees (for example verification of the official patches) and does not require to "run" the code.

D. Kroening's CBMC [5] has been used for many practical applications. A good overview over the applied fields can be found under the following link [6].

7. CONCLUSION

In this paper we combined state of the art model checking with theoretical work on Quantitative Information Flow, to provide a powerful tool for the analysis of leakage of information. We demonstrated not only that CVE reported vulnerabilities such as for the Linux kernel can be analysed with a level of scalability and precision able to find real security vulnerabilities, but that it is also possible to prove whether the official patches fix the problem. We argued that leaks are not synonymous of a security breach and hence a quantitative framework is better equipped than a qualitative one to determine when an information leak represents a security threat.

We see this work as a significant step in the application of academic research on information flow analysis to real-world problems in systems software.

Acknowledgment We thank Peter O'Hearn for helpful comments on the paper. This research was funded by EPSRC, grant EP/F023766/1, with title "Model Checking and Program Analysis for Quantifying Interference".

8. REFERENCES

[1] Michael Backes and Boris Köpf and Andrey Rybalchenko: Automatic Discovery and Quantification of Information Leaks. Proc. 30th IEEE Symposium on Security and Privacy (S&P '09)

[2] Barthe, Gilles and D'Argenio, Pedro R. and Rezk, Tamara: Secure Information Flow by Self-Composition. CSFW '04: Proceedings of the 17th IEEE workshop on Computer Security Foundations.

[3] David Clark, Sebastian Hunt, Pasquale Malacaria: A static analysis for quantifying information flow in a simple imperative language. Journal of Computer Security, Volume 15, Number 3 / 2007.

[4] David Clark, Sebastian Hunt, and Pasquale Malacaria: Quantitative information flow, relations and polymorphic types. Journal of Logic and Computation, Special Issue on Lambda-calculus, type theory and natural language, 18(2):181-199, 2005.

[5] Clarke, Edmund, and Kroening, Daniel, and Lerda, Flavio: A Tool for Checking ANSI-C Programs. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004). Springer, 168–176, Volume 2988

[6] http://www.cprover.org/cbmc/applications.shtml – Checked 17 June 2010.

[7] Joseph A. Goguen, Jose Meseguer: Security Policies and Security Models. IEEE Symposium on Security and Privacy 1982: 11-20

[8] Jonathan Heusser and Pasquale Malacaria: Applied Quantitative Information Flow and Statistical Databases. Formal Aspects in Security and Trust 2009: 96-110

[9] Boris Köpf and Andrey Rybalchenko: Approximation and randomization for quantitative information-flow analysis. In Proceedings CST 2010

[10] Landauer, J., and Redmond, T.: A Lattice of Information. In Proc. of the IEEE Computer Security Foundations Workshop. IEEE Computer Society Press, 1993.

[11] Pasquale Malacaria: Assessing security threats of looping constructs. Proc. ACM Symposium on Principles of Programming Language, 2007.

[12] Pasquale Malacaria, Han Chen: Lagrange multipliers and maximum information leakage in different observational models. PLAS 2008: 135-146

[13] Pasquale Malacaria and Jonathan Heusser: Information Theory and Security: Quantitative Information Flow. In Formal Methods for Quantitative Aspects of Programming Languages, LNCS, Springer Verlag, 2010

[14] Stephen McCamant, Michael D. Ernst: Quantitative information flow as network flow capacity. PLDI 2008: 193-205 MIT Department of Electrical Engineering and Computer Science, Ph.D., Cambridge, MA, 2008.

[15] James Newsome, Stephen McCamant, Dawn Song: Measuring channel capacity to distinguish undue influence. PLAS 2009: 73-85

[16] Benjamin Schwarz, Hao Chen, David Wagner, Jeremy Lin, Wei Tu, Geoff Morrison, Jacob West: Model Checking An Entire Linux Distribution for Security Violations. ACSAC 2005: 13-22

[17] Pasareanu, Corina S. and Dwyer, Matthew B. and Huth, Michael: Assume-Guarantee Model Checking of Software: A Comparative Case Study. Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking,1999

[18] T. Terauchi and A. Aiken. Secure information flow as a safety problem: In SAS, volume 3672 of LNCS, pages 352–367, 2005.
[19] Hirotoshi Yasuoka and Tachio Terauchi Quantitative information flow - verification hardness and possibilities. In Proceedings CSF 2010.

Analyzing and Improving Linux Kernel Memory Protection: A Model Checking Approach

Siarhei Liakh
North Carolina State University
sliakh@ncsu.edu

Michael Grace
North Carolina State University
mcgrace@ncsu.edu

Xuxian Jiang
North Carolina State University
jiang@cs.ncsu.edu

ABSTRACT

Code injection continues to pose a serious threat to computer systems. Among existing solutions, $W \oplus X$ is a notable approach to prevent the execution of injected code. In this paper, we focus on the Linux kernel memory protection and systematically check for possible $W \oplus X$ violations in the Linux kernel design and implementation. In particular, we have developed a Murphi-based abstract model and used it to discover several serious shortcomings in the current Linux kernel that violate the $W \oplus X$ property. We have confirmed with the Linux community the presence of these problems and accordingly developed five Linux kernel patches. (Four of them are in the process of being integrated into the mainline Linux kernel.) Our evaluation with these patches indicate that they involve only minimal changes to the existing code base and incur negligible performance overhead.

1. INTRODUCTION

Despite years of research, code injection attacks continue to be one of the major ways of computer break-ins and malware propagation [21]. Specifically, a code injection attack is a method whereby an attacker inserts malicious code into a running process and transfers execution to the malicious code (e.g., by hijacking its control flow). After that, the attacker can gain control of a running process and carry out other malicious activities, including the installation of bot programs for remote control and the modification of system files to allow for unauthorized access, etc.

There exist a variety of solutions [15, 25, 28, 30, 33] to deal with code injection attacks. Among the most notable, $W \oplus X$ [1] is a scheme that has been proposed to counter code injection attacks. In essence, $W \oplus X$ enforces the following property, "a given memory page will never be both writable and executable at the same time." The basic premise behind it is that if a page cannot be written to and later executed from, code injection becomes hard, if not impossible, to launch. Due to its effectiveness in defending against code injection attacks, since its proposition, $W \oplus X$ has been widely adopted in commodity OSs (e.g., Windows and Linux). Hardware vendors such as Intel and AMD also follow up this scheme by providing necessary hardware support (in the form of NX support [9]) to facilitate the $W \oplus X$ enforcement.

From the OS kernel perspective, establishing and maintaining the $W \oplus X$ property requires a sound design. In this paper, we look into the Linux kernel and analyze the way it takes to protect its own kernel memory. This is important as the Linux kernel is typically a part of trusted computing base (TCB) in existing solutions to defend against code injection attacks. In our analysis, we took a model checking approach so that we can take advantage of its power to rigorously examine the soundness and completeness of $W \oplus X$ enforcement in Linux kernel. More specifically, we first build a model of Linux kernel memory management subsystem and then apply model checking to verify the $W \oplus X$ property. In case of violation, model checking has the unique advantage in accurately pinpointing potential problems in the current Linux kernel design and implementation.

We have successfully developed a Murphi [10]-based abstract model to analyze linux kernel memory management. Based on our modeling, we were surprised to discover several issues related to Linux kernel memory management: (1) First, the current Linux kernel does not strictly separate the kernel code and kernel data, immediately leading to $W \oplus X$ violation. (2) Second, as part of its implementation, Linux kernel promotes creation of multiple virtual aliases, potentially with conflicting permissions, for the same physical memory page, leading to exploitable scenarios for kernel data execution or kernel code modification.

We have confirmed with the Linux kernel community the presence of these issues. We have also accordingly developed kernel patches to fix these problems and these patches [17, 18, 19, 20] are in the process of being integrated into mainstream Linux kernel.[2] Our evaluation indicates that these patches are compatible with existing Linux kernel code base and contain minimum modifications to the existing interfaces that manage kernel memory. We also observe that

[1]Strictly speaking, the property is $\neg(W \wedge X)$, but we chose to use the traditional $W \oplus X$ notation to emphasize mutual exclusivity of write and execute access.

[2]For the convenience of kernel patch debugging and adoption, we have developed five smaller patches in total (Section 4). Four of them are being tested for final integration into mainstream Linux kernel and the remaining one is still being internally assessed for suitability in mainline Linux kernel.

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

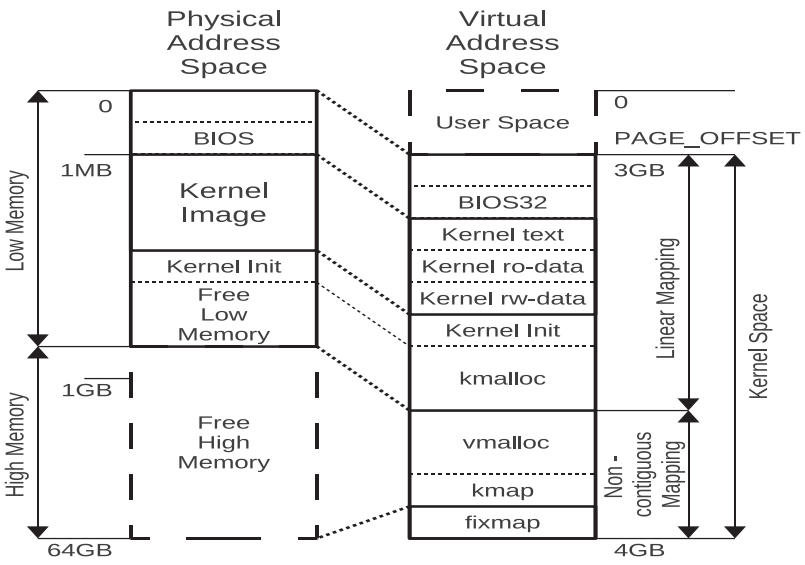


Figure 1: Typical Linux Memory Mapping

these patches impose virtually no performance overhead despite a moderate increase in memory consumption.

The rest of the paper is structured as follows. We start with Linux kernel modeling in Section 2 and present the model checking results in Section 3. Then we present and evaluate our solution in Section 4 and Section 5, respectively. After that, we examine limitations of our solution and suggest possible improvements in Section 6. Finally, we discuss related work in Section 7 and conclude our paper in Section 8.

2. MODELING AND DESIGN

2.1 Murphi Background

Our abstract modeling of Linux kernel memory protection is based on Murphi [10], which is both a language and a tool for model verification with explicit state enumeration. To use Murphi, we need to write a Finite State Machine (FSM) description, which will be taken as an input by Murphi to produce an executable. The executable, once started, will perform the model verification task and produce the output detailing the verification results.

In Murphi, a finite state machine description consists of three parts: a set of *states*, a set of *transition rules* and an *initial state*.

- The set of *states* is implicitly defined through the declaration of global variables. Each combination of values of each variable naturally produces a unique state of the system. Note that not all of these potential combinations are reachable in the FSM.
- The transitions between the states of FSM are defined through a set of *transition rules*, each consisting of two parts, a *guard* and an *action*. A guard is a logic expression that determines the conditions under which an action can be taken. An action is a set of instructions to manipulate the global variables, thus transitioning the FSM from one state to another. An action is performed if, and only if the corresponding guard is evaluated to be *TRUE*.
- An *initial state* is defined by a special rule with an action which is executed only once, right before the FSM state exploration begins. This allows to explicitly define an initial state of the FSM.

For model verification purposes, Murphi also adds to the FSM a fourth component: *invariants*. The invariants are a set of logic expressions which define a set of *safe* states of FSM. A state is only considered *safe* iff all invariants evaluate to *TRUE* in this state. With that, model verification is performed by exploring all reachable states, starting with a given initial state. The state space exploration is performed by applying all possible rules to all states that have already been reached with standard algorithms such as depth- or breadth-first search. For each newly discovered reachable state Murphi evaluates the invariants. Should any invariant evaluate *FALSE*, the system reports an error and prints out a set of states and transitions that led to the unsafe state. The state exploration continues forward only when all invariants evaluate to *TRUE* for the newly discovered state. In this work we apply model checking twice: one in detecting the $W \oplus X$ violation in the current Linux kernel (Section 3) and another in validating our suggested solution (Section 4).

2.2 Linux Kernel Memory Model

For proper modeling, it is important to understand how physical memory is being organized and mapped in Linux. On a typical 32-bit Linux system with the paging-based virtual memory enabled, the 4GB virtual memory space is split (Figure 1) between user space (bottom 3GB) and kernel space (top 1GB). In other words, the kernel space starts at address $PAGE\_OFFSET = 0xc0000000$ ($3GB$) and stretches up to the end of virtual address space $0xffffffff$ ($4GB-1$). We also notice that on an i386 architecture, physical memory is typically split onto two regions: *low memory*

and *high memory*. The low memory region starts at address $0x00000000$ and ends at around $800MB$.[3] In Linux, this region is mapped directly into the kernel space, starting at *PAGE_OFFSET* and called *linear mapping*. Such linear mapping allows for simple translation between the physical and virtual addresses for all pages within this region. Specifically, the virtual address of any location within low memory can be obtained by adding *PAGE_OFFSET* value to the physical address. The reverse translation is performed by simply subtracting *PAGE_OFFSET* from the virtual address, without the need for page table lookup.

We point out that the linear mapping is established at kernel startup and persists all the way through system shutdown [5, 13, 23]. The virtual address space within this region is used to contain the following parts of the Linux kernel: BIOS32 services, kernel text, kernel read-only data, kernel read-write data and dynamic memory allocations that require contiguous physical pages (e.g., through *kmalloc()* call). Kernel initialization routines are also loaded here and later released as free memory.

Assuming the total amount of physical memory installed in the system is greater than the maximum size of low memory for the current kernel configuration, the high memory lays in the physical address space immediately following the low memory. However, unlike low memory, high memory is only mapped into kernel space when needed. There are two basic mechanisms through which this memory is mapped: *vmalloc()* and *kmap()*. The major difference between the two is that the former is used for long-term allocation of non-contiguous physical memory into contiguous virtual address space (e.g., for loadable kernel module allocation), while the latter is used strictly for short-term access to physical pages located in the high memory. Another difference is that vmalloc() may allocate pages from either high or low memory, while kmap() is strictly used for high memory only. Also, when vmalloc() allocates a page from low memory, it creates an *alias*: the same physical page will be mapped into the kernel virtual address space twice, one time in linear area and another one in vmalloc() area. The problem with such aliasing is that memory management subsystem has to ensure consistency of page attributes between all of the aliases. In the context of this work, both vmalloc() and kmap() areas play the same role: mapping non-contiguous physical pages into the kernel address space. Therefore, we treat them as the same in our model.

There is another memory area in Linux kernel called *fixmap*, which is located at the very end of the address space and mainly used by kernel to establish reserved, pre-defined fixed mappings such as PCI control registers and other memory-mapped services. Similar to the linear mapping, the fixmap region is established at kernel startup and persists all the way until system shutdown. Since this area does not represent an unique type of mapping that is distinct from the ones described above, we do not explicitly include it in the model.

Note that all modern multi-tasking operating systems rely on hardware support to provide virtual memory (and the $W \oplus X$ is enforced only when the virtual memory is enabled). In order to accurately model hardware support, we therefore include an abstraction of hardware memory subsystem in our model. Specifically, our model covers the mapping from physical memory to virtual memory as well as the related page tables and access flags. Our model is based on a single-level abstraction of the i386 family paging mechanism [2]. In other words, the mapping of virtual addresses to physical pages is modeled by a flat page table array (*pg_table_t*).[4] Each entry (*pte_t*) of that array corresponds to a virtual page (if *mapped*) and holds related attributes such as physical address (*addr*) it is currently mapped to and access permissions (*prot*). In our model we use *pgprot_t* type for page attribute tracking. This type defines two page access attributes: "write" and "execute". Since kernel pages are not swappable and all mapped pages are always readable, we do not model the "read" and "present" flags. Also, since this model is only concerned with kernel space, we do not model the *user/supervisor* flag. For the physical memory, due to the aliasing, we model them as an array (*frame_table_t*) of physical memory frames, with each physical page (*frame_t*) holding a reference count (*reference_count* or the number of mapped virtual pages) and a most permissive set (*prot*) of the attributes derived from all the virtual aliases pointing to it. As a result, the global definition of the page table array and the physical memory array define the possible *states* in our model.

```
-- page protection attributes
pgprot_t: record
  w: boolean;
  x: boolean;
end;

-- page table entry
-- (mem_index is the memory frame number: [1..
    mem_size])
pte_t: record
  mapped: boolean;
  prot: pgprot_t;
  addr: mem_index;
end;

-- single-level page table
-- (page_index is the page table index: [1..
    pt_size])
pg_table_t: array[page_index] of pte_t;

-- physical memory frames
frame_t: record
    reference_count: 0..pt_size;
    prot: pgprot_t;
end;
frame_table_t: array[mem_index] of frame_t;
```

Based on the above abstraction, our model then captures the specifics of the initial state of Linux kernel. As mentioned earlier and shown in Figure 1, the initialization of Linux kernel memory involves three main parts: kernel linear mapping, static kernel image mapping, and mapping of BIOS32 services. Accordingly, our model represents them by establishing three basic properties as part of the model's *initial state*: ***S1*** *- linear mapping*, ***S2*** *- static kernel mapping*, and ***S3*** *- BIOS32*. The details about them can be found in Appendix A.

After that, we further obtain the transition rules in our model. In particular, based on the Linux kernel source code and our domain knowledge, we identify and extract a number of kernel function routines or application program in-

[3]The exact value depends on the physical memory size and other compile-time and run-time kernel configuration.

[4]Multiple levels of page tables are not necessary as they only allow to map large numbers of pages more efficiently and do not introduce any additional qualities related to this work. The same also stands true for the "large pages" introduced by Page Size Extension (PSE) technology [2].

terfaces (APIs) that are used to affect the kernel memory mapping. Some of them are:

- *map_vm_area()* maps a physical page to a virtual address.
- *static_protections()* ensures that pre-defined areas of kernel's virtual address space always have correct attributes (kernel code should stay executable, kernel data readable and so on.)
- *cpa_process_alias()* checks all mapped aliases for a given physical page frame and updates them as necessary.
- *__change_page_attr_set_clr()* receives a block request for memory attribute change and translates it into a series of attribute and alias check calls for each individual page.
- *__change_page_attr()* executes attribute change for the individual pages.

To represent them, we derive a set of basic rules to capture their behavior especially when they perform memory mapping, re-mapping or change memory page attributes. By doing so, we avoid the need of understanding specific memory use-cases such as kernel module loading or unloading (as it is already captured with these APIs). In our model, we have three key transition rules and use them in our Murphi-based FSM description.

- ***T1*** - *Set*: This transition rule sets W and/or X flags for a given page table entry.
- ***T2*** - *Clear*: This transition rule clears W and/or X flags for a given page table entry.
- ***T3*** - *Map*: This transition rule changes the mapping between a physical frame and a virtual page.

```
ruleset i: page_index do
  ruleset x: boolean do
    ruleset w: boolean do

      -- set W and/or X for virtual page i
      rule "T1: Set"
        true ==> begin
          set_mem_perm(i, 1, w, x);
        end;

      -- clear W and/or X for virtual page i
      rule "T2: Clear"
        true ==> begin
          clr_mem_perm(i, 1, w, x);
        end;

      -- map page i to physical address pa with
          prot attributes
      ruleset pa: mem_index do
        rule "T3: Map"
          true ==> begin
            map_vm_area(i, pa, prot);
          end;
      end;

    end;
  end;
end;
```

3. ANALYSIS

After obtaining the abstract model of Linux kernel, we further define the invariants to analyze possible Linux kernel states. Since our focus in this work is on the $W \oplus X$ enforcement, we established the following properties through invariants in the model:

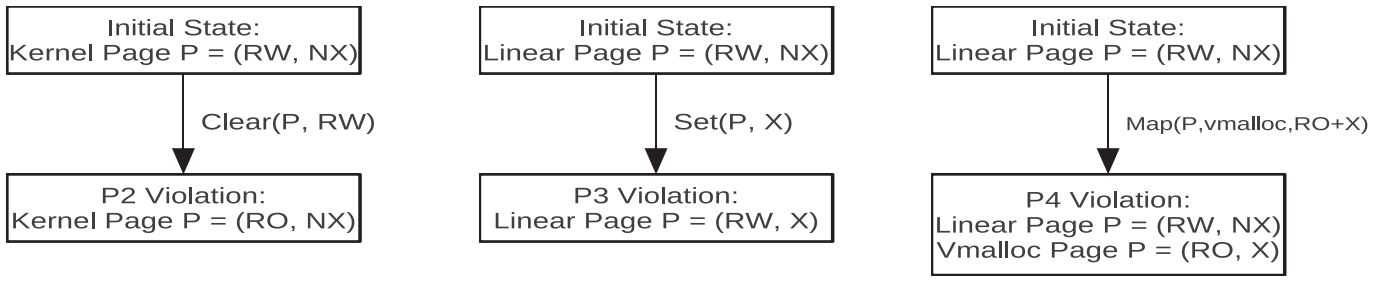
- ***P1***: Kernel code should always be executable and read-only.
- ***P2***: Kernel data should always be non-executable, the read-only kernel data should remain read-only, and read-write kernel data should always be writable.
- ***P3***: No page will be writable and executable at the same time in order to not violate $W \oplus X$.
- ***P4***: All virtual aliases of each physical page should have consistent access permissions.

```
invariant "P1: Kernel ROX Code"
  true -> kernel_code_rox() = true;

invariant "P2: Kernel RO data"
  true -> kernel_rodata_ronx() = true;

invariant "P2: Kernel RW data"
  true -> kernel_rwdata_rwnx() = true;

invariant "P3: W xor X"
  true -> w_and_x() = false;

invariant "P4: Alias consistency"
  true -> page_alias_matching() = true;
```

More specifically, the P1 invariant is necessary to keep kernel code executable because non-executable kernel code will lead to an immediate system crash. The P2 invariant ensures that read-only kernel data that holds constants cannot be modified and that read-write data is always accessible. It also ensures that static kernel data cannot be executed. The P3 invariant states that any given page cannot be writable and executable at the same time. This property is necessary to prevent code injection and is focal point of this work. The P4 invariant is necessary to prevent code injection by accessing an alias which is mapped into a different virtual address with different access permissions.

With invariants in place, the model checker is able to provide us with examples of possible transitions if they lead to an unsafe state that violates $W \oplus X$ policy. Our experience with the model checker indicates that there is no P1 violation in the current Linux design and implementation. However, it reports violations for all other three invariants (Figure 2).

P2 violation The violation of P2 arises when kernel read-write data region is set as read-only. The problem stems from the fact that *static_protections()* does not check for the correctness of new access flags set for the kernel *read-write data* region. While this issue does not directly allow for code injection, setting read-write data (Figure 2(a)) as read-only provides a vector for a denial of service attack. The reason is that the kernel assumes that its read-write data is always writable and is not equipped to handle this situation. This issue has been confirmed by a kernel crash, which immediately follows the call of *set_pages_ro()* for the read-write data region. During the investigation of this issue, we have also identified and confirmed a kernel bug that



(a) P2 Violation: Kernel Read/Write Data (b) P3 Violation: W and X (c) P4 Violation: inconsistent aliases

Figure 2: Property Violations

can be used to apply improper page attributes to a memory region when large (2MB) pages are used in the kernel space (Section 4).

P3 violation The violation of P3 happens in two scenarios. The first one occurs in the initial state. More specifically, the way the BIOS32 is mapped into the kernel space during the initialization directly contradicts the P3 invariant. Note that the BIOS32 services contain executable code and, therefore, should be set as read-only. However, the Linux kernel simply indiscriminately maps the whole BIOS region into the kernel address space as writable and executable. While the actual BIOS code is typically stored in ROM and cannot be overwritten, such mapping still provides an opportunity for data execution or code modification. In a typical page table dump of the latest Linux kernel (version 2.6.33) shown in Figure 3(a), this violation manifests itself as a set of $RW + X$ pages within the range *0xc0000000 - 0xc2000000*.

The second scenario is related to the original memory management interface that allows for pages to be mapped as writable and executable at the same time, thus violating P3 (Figure 2(b)). The source of the problem is the absence of any access permission verification system for memory pages outside the static kernel image in the default memory management interface, such as when a kernel module is being loaded. To confirm this violation, we loaded several modules and inspected the vmalloc() area of the page tables for $W \oplus X$ violations. We found that Linux kernel indeed does not enforce $W \oplus X$ for mappings in the non-contiguous memory region (Figure 3(a)).

P4 violation This violation of P4 happens when a page from *low memory* region is being mapped into vmalloc() area as executable. This would typically happen during the memory allocation for a loadable kernel module on a system that does not have any *high memory* available for allocation. A module loader would use vmalloc() to load all module sections, including code with execute permissions. However, the physical page is already mapped once in the linear mapping space with $RW + NX$ permissions, creating an opportunity for code injection. We were able to re-create this behavior consistently by loading Linux kernel in a virtual machine with only 128MB of RAM, forcing Linux to allocate all pages from low memory and thus creating aliases for each of vmalloc() allocations.

4. IMPROVEMENTS AND PROTOTYPE

Based on the findings described in the previous section, we propose a few improvements to the Linux kernel. Our improvement involves the changes to the initial states (S1, S2, and S3) as well as the transition rules (T1, T2, and T3) such that no unsafe states will be reached from the revised FSM description. For debugging and verification purposes, we have developed five self-contained and independent patches (in total 703 source code of lines) to the Linux kernel, four of which have been submitted to LKML [17, 18, 19, 20] and are currently in the process of being integrated into the mainline kernel. The remaining one is still being assessed for suitability in the mainline kernel.

In the development of these patches, based on the counter-examples reported from Murphi, we revised the memory management subsystem in Linux. Interestingly, we found that there are two distinct levels of abstraction interfaces to manage Linux kernel memory. The low-level interface is primarily tasked with direct manipulation of page table entries, and the high-level interface is used to abstract page table layout and provide functions like *set_memory_nx()*. At first glance, the low-level interface seems a better target to address previous violations. However, the following significant shortcomings in its design complicate such approach. Therefore, we chose the high-level interface as a target for our solution.

- First, the low-level functions (e.g., *pte_mkexec()*) are intended as simple data manipulation primitives which cannot fail. Our experience indicates that any deviation from this assumption would yield unpredictable results, as the current kernel is not yet designed to handle such failures gracefully.
- Second, these functions operate on instances of *pte_t* type, which by itself does not guarantee that the instance is in fact part of active or future page tables. This, in turn, means that we would either need to check the address of each *pte_t* in question, or enforce $W \oplus X$ on all instances of *pte_t*. Neither of these options is desirable: the former creates a significant performance overhead while the latter one introduces unwanted side effects into all intermediate transformations of *pte_t* instances.

4.1 Fixing P2 Violation: Preserving the *write* access on kernel read-write data

Our first patch fixes the violation of P2 invariant. More specifically, as discussed earlier, this problem is caused by the inability of *static_protections()* to preserve the *write* access to kernel *read-write data* (including the *BSS* section). The fix is a straightforward one. However, in the process

```
---[ Kernel Mapping ]---
0xc0000000-0xc0200000          2M     RW             GLB x  pte
0xc0200000-0xc0600000          4M     ro         PSE GLB x  pmd
0xc0600000-0xc0843000       2316K     ro             GLB x  pte
0xc0843000-0xc0a00000       1780K     RW             GLB x  pte
0xc0a00000-0xf7800000        878M     RW         PSE GLB NX pmd
0xf7800000-0xf79fe000       2040K     RW             GLB NX pte
0xf79fe000-0xf7a00000          8K                           pte
0xf7a00000-0xf8000000          6M                           pmd
0xf8000000-0xf81fe000       2040K                           pte
---[ vmalloc() Area ]---
0xf81fe000-0xf81ff000          4K     RW     PCD     GLB NX pte
0xf81ff000-0xf8200000          4K                           pte
                          [ . . . ]
0xf8247000-0xf824a000         12K     RW             GLB x  pte
0xf824a000-0xf824c000          8K                           pte
0xf824c000-0xf824d000          4K     RW             GLB x  pte
0xf824d000-0xf824f000          8K                           pte
0xf824f000-0xf8274000        148K     RW             GLB NX pte
0xf8274000-0xf8276000          8K                           pte
0xf8276000-0xf8278000          8K     RW             GLB x  pte
0xf8278000-0xf827a000          8K                           pte
0xf827a000-0xf827b000          4K     RW             GLB x  pte
0xf827b000-0xf827d000          8K                           pte
0xf827d000-0xf8296000        100K     RW             GLB NX pte
0xf8296000-0xf8298000          8K                           pte
                        [ . . . ]
```

(a) Vanilla Kernel

```
---[ Kernel Mapping ]---
0xc0000000-0xc00fb000       1004K     RW             GLB NX pte
0xc00fb000-0xc00fd000          8K     ro             GLB x  pte
0xc00fd000-0xc0200000       1036K     RW             GLB NX pte
0xc0200000-0xc0600000          4M     ro         PSE GLB x  pmd
0xc0600000-0xc068e000        568K     ro             GLB x  pte
0xc068e000-0xc0844000       1752K     ro             GLB NX pte
0xc0844000-0xc0a00000       1776K     RW             GLB NX pte
0xc0a00000-0xf7800000        878M     RW         PSE GLB NX pmd
0xf7800000-0xf79fe000       2040K     RW             GLB NX pte
0xf79fe000-0xf7a00000          8K                           pte
0xf7a00000-0xf8000000          6M                           pmd
0xf8000000-0xf81fe000       2040K                           pte
---[ vmalloc() Area ]---
0xf81fe000-0xf81ff000          4K     RW     PCD     GLB NX pte
0xf81ff000-0xf8200000          4K                           pte
                          [ . . . ]
0xf821a000-0xf821d000         12K     ro             GLB x  pte
0xf821d000-0xf821e000          4K     ro             GLB NX pte
0xf821e000-0xf8220000          8K     RW             GLB NX pte
0xf8220000-0xf8222000          8K                           pte
0xf8222000-0xf8223000          4K     RW     PCD     GLB NX pte
0xf8223000-0xf8227000         16K                           pte
0xf8227000-0xf8228000          4K     ro             GLB x  pte
0xf8228000-0xf8229000          4K     ro             GLB NX pte
0xf8229000-0xf822b000          8K     RW             GLB NX pte
                        [ . . . ]
```

(b) Patched Kernel

Figure 3: Dumping Kernel Page Tables (kernel version 2.6.33**)**

of re-mapping our model back to the original Linux kernel source code, we discovered another implementation bug in the function routine *try_preserve_large_pages()*. Specifically, this function incorrectly processes access permission change requests for areas that start on a boundary of a large page (i.e., $2M$), but are smaller than the page itself. This leads to a possibility of setting improper access flags to the memory area located within the same large page, but immediately after the requested one. Specifically, this problem manifested itself by kernel read-write data becoming read-only simply because it was initially mapped within the same large page as kernel's read-only data. Accordingly, we propose two changes in the patch (that affects the file *arch/x86/mm/pageattr.c*): one is to allow *static_protections()* to preserve the *write* access for kernel's read-write data area and another one is to verify each small page within the large page for access flag compatibility [17].

4.2 Fixing P3 Violation: Removing mixed pages in kernel space

Our next three patches address P3 violation, namely the presence of mixed code and data pages in kernel space. Based on our model checking results, our investigation maps the related transition rules that lead to an unsafe state back to the involved kernel routines. By doing so, we are able to identify three distinct sources: BIOS32, loadable kernel modules (LKMs), and static kernel image management.

BIOS32 As discussed in Section 3, the current Linux kernel improperly maps the entire BIOS area into the kernel space as $RW + X$. (Note the BIOS code itself is typically located in read-only memory or ROM.) To resolve this issue, we implemented a patch that dynamically maps BIOS32 services into the kernel space. Based on related BIOS32 documents [4] and [26], it requires at most two pages to be executable per BIOS32 service, and none of them are expected to be writable. As such, a dynamic service mapping of BIOS32 services can be established at the time of service discovery, and with appropriate (and $W \oplus X$-compliant) access permissions. As part of our patch, we added the BIOS32 service mapper to *pci/pcbios.c* and removed unnecessary protection for the area of physical memory under 2MB from *arch/x86/mm/pageattr.c*. Also, we revised the file *mm/init_32.c* to properly report kernel text addresses [18] because of the BIOS32 changes.

LKMs The second source of violating $W \oplus X$ is located in the support of LKMs. In particular, since all vmalloc() allocations default to "data" access mode ($RW + NX$), the only source of mixed pages in this area is LKMs as their code is explicitly marked as "executable". More specifically, dynamic kernel linker allocates each loadable module in two parts: module init and module core. The init part of the module will be discarded after initialization, while core will stay resident in the kernel. In order to minimize a module's footprint, the linker chooses the minimum amount of spacing necessary between each of the module's sections - just enough to accommodate necessary section alignment, which introduces mixed kernel pages. Accordingly, our patch allocates module sections in three groups: text, read-only data, read-write data and further adjusts the linker to align each of the groups on a page boundary, ensuring that each page contains only sections from the same group. Next, our patch assigns a set of appropriate access permissions to all pages of each group as follows: read-only for text and read-only data, non-executable for read-only data and read-write data. As this patch will inevitably introduce additional memory consumption (Section 5), we create a compile-time option (i.e., *CONFIG_DEBUG_SET_MODULE_RONX*) to turn on or off the functionality of this patch as needed [20].

Static kernel image Similar to the support of LKMs, our next patch includes the code to split all sections of the static kernel image into three groups: text, read-only data, and read-write data. Specifically, our patch addresses necessary group alignment by modifying the related linker script (*kernel/vmlinux.lds.S*) and assigns proper access permissions to the pages of each group at the end of kernel initialization (*mm/init.c* [19]). The functionality of this patch is always enabled.

4.3 Fixing P4 Violation: Disallowing memory aliasing with permission conflicts

The remaining patch addresses the P4 violation. In particular, as we have mentioned in Section 2.2, memory alias han-

dling is being implemented in the kernel in *cpa_process_alias()* function. However, the way it handles page aliases specifically excludes the case of an NX update.

```
/* No alias checking for _NX bit modifications */
checkalias = (pgprot_val(mask_set) | pgprot_val(
    mask_clr)) != _PAGE_NX;
```

To enforce $W \oplus X$, our patch needs to modify the alias handling routine such that it will propagate changes of all access flags to all aliases. This can be achieved by setting *checkalias = 1* for all page attribute modifications.

```
static inline pgprot_t
  process_WxorX_violation(pgprot_t prot,
     unsigned long address, unsigned long pfn)
{
  /*
   * We can Oops or Panic here if needed. But for
        now we just print out an error message.
   */
  printk(KERN_ERR "(W xor X) violation: " "VA=0x%
      lx, PFN=0x%lx", address, pfn);
  /* Set NX, just in case */
  pgprot_val(prot) |= _PAGE_NX;
  return prot;
}
```

In addition, our patch implemented a helper routine *process_WxorX_violation()* (a part of *mm/pageattr.c*) for the strict enforcement of $W \oplus X$ property. Particularly, this function will be called for each page that is about to violate $W \oplus X$ property, before the new protection attributes can be applied. Because of the presence of aliasing, this routine is called for any inconsistency of memory protection attributes in aliases as well. If there are conflicting attributes, this function will by default set NX flag for all related pages and logs an error message detailing the associated virtual and physical addresses.

5. EVALUATION

Our improvements aim to make the Linux kernel conform to the $W \oplus X$ property by guaranteeing exclusivity of write and execute page access. This means that all of the kernel code that follows the interfaces in place will automatically be compliant with $W \oplus X$ without additional modifications. To rigorously verify the $W \oplus X$ compliance, we revise the previous FSM description to reflect our patches (Section 4). Also, in order to avoid unnecessary state explosion, we chose a minimal configuration where we only modeled a system that contains one virtual page of each type: kernel text, kernel read-only data, kernel read-write data, linear mapping. To allow for variability, we use two pages in non-contiguous mappings, and model the physical memory with one more page frame than the total size of the virtual address space. In addition, the model contained a proposed memory interface, a full set of rules, and invariants to establish and monitor the $W \oplus X$ property. The model checker examined 27942 states and 7823760 rules without detecting any violations.

In order to further confirm the validity of our approach in a real system, we used a minimal Ubuntu Server 8.04.4 LTS [32] system that runs the latest vanilla Linux 2.6.33 kernel [16]. Note the vanilla Linux kernel has been compiled with Generic Ubuntu configuration, and then booted and inspected. In Figure 3(a), we show the virtual memory layout in the vanilla Linux kernel. It shows that while all necessary elements of code and data separation are indeed present, they are not applied in a consistent manner. Specifically, RO and NX flags are used sparsely. As shown from the detailed kernel page table dump (Figure 3(a)), it fails to establish $W \oplus X$ property.

In comparison, we applied our patch set to the same kernel and repeated the inspection. A clear difference can be observed on Figure 3(b): we have successfully eliminated all pages with mixed access. Specifically, the following changes are noteworthy:

- a 2Mb area between *0xc0000000* and *0xc0200000* is now marked as $RW + NX$, except for two $RO + X$ pages *0xc00fb000* - *0xc00fd000* reserved for BIOS32 services.
- Static Kernel image (*0xc02000000* - *0xc08a3000*) is clearly partitioned in three sections: $RO + X$ code (*0xc02000000* - *0xc068e000*), $RO+NX$ read-only data (*0xc08a3000* - *0xc0844000*), and $RW+NX$ read-write data (*0xc0844000* - *0xc08a3000*).
- Loadable Kernel Modules (see addresses *0xf821a000* - *0xf8220000* and *0xf8227000* - *0xf822b000*) are now clearly split into three parts: $RO+X$ code, $RO+NX$ read-only data, and $RW + NX$ read-write data.

Performance and Memory Overhead To evaluate the performance overhead introduced, we measured its runtime overhead with three tasks: UnixBench 5.1.2 [31] (index test group, SMP and Uniprocessor configurations), Linux kernel compilation, and compression time of 10GB random data stream. All tests have been performed on Ubuntu Server 8.04.4 LTS with Linux 2.6.33 compiled for i386 architecture in standard Ubuntu Server configuration except for PAE enabled, and XEN disabled. Our test platform is a Gigabyte MA78G-DS3HP system (AMD RS780/SB700 chipset) with dual-core AMD Athlon 4850e processor (family 15, model 107, stepping 2) and 8GB of PC2-6400 RAM in dual-channel configuration (4x2GB, CL5). The results are shown in Table 1.

Our results indicate that our patch set does not affect overall system performance in a measurable way. In particular, the first four patches (related to static kernel image, LKM, and BIOS32) do not introduce any additional performance overhead as all additional work are mainly performed at compile-time. Though there is a slight overhead incurred during the kernel/module initialization, it does not affect the runtime performance after initialization. Among the five patches, the only patch that involves run-time penalty is the compliance checking of $W \oplus X$ for each kernel page table update (i.e., in the helper routine *process_WxorX_violation()*).

Next, we evaluate the memory overhead introduced into the kernel by our patches. We first investigate the difference in the size of the static kernel image and memory allocated to individual modules. As can be seen on Figure 4, the size of the static kernel image has increased from $6793KB$ ($4660KB$ text + $2133KB$ data) to $6796KB$ ($4664KB$ text + $2132KB$ data). This increase constitutes a mere 0.04%, and is thus not significant.

We also used the same system to load 44 kernel modules of varying sizes. Due to the fact that our patch set affects the layout of different module sections, the total size of these 44 modules (reported by running the *lsmod* command) is increased from $1,265,502B$ to $1,493,138B$ (an increase of 22.01%). Note that the module sizes reported by *lsmod*

```
fixmap  : 0xfff1e000 - 0xfffff000   ( 900 kB)
pkmap   : 0xffa00000 - 0xffc00000   (2048 kB)
vmalloc : 0xf81fe000 - 0xff9fe000   ( 120 MB)
lowmem  : 0xc0000000 - 0xf79fe000   ( 889 MB)
  .init : 0xc08a3000 - 0xc0916000   ( 460 kB)
  .data : 0xc068d32c - 0xc08a29e8   (2133 kB)
  .text : 0xc0200000 - 0xc068d32c   (4660 kB)
```

(a) Vanilla Kernel

```
fixmap  : 0xfff1e000 - 0xfffff000   ( 900 kB)
pkmap   : 0xffa00000 - 0xffc00000   (2048 kB)
vmalloc : 0xf81fe000 - 0xff9fe000   ( 120 MB)
lowmem  : 0xc0000000 - 0xf79fe000   ( 889 MB)
  .init : 0xc08a3000 - 0xc0916000   ( 460 kB)
  .data : 0xc068e000 - 0xc08a3000   (2132 kB)
  .text : 0xc0200000 - 0xc068e000   (4664 kB)
```

(b) Patched Kernel

Figure 4: Kernel Virtual Memory Layout

Benchmark	Vanilla	Patched	Overhead%
UnixBench UP (index)	737.88	741.8	0.53%
UnixBench SMP (index)	1317.13	1310.88	-0.47%
Kernel Compilation (seconds)	1712	1725	0.76%
Gzip test (seconds)	2890	2873	-0.59%

Table 1: Run-time $W \oplus X$ enforcement overhead

could be misleading, as they do *not* reflect the fact that kernel allocates memory for modules at the page granularity. This means that the true measure of module memory consumption should be the memory (in terms of pages) allocated to the module, not the module size. With that, if we can take the whole-page allocation into account, the memory overhead is increased from 322 pages (size of $4K$) to 382 pages (an increase of 22.17%). We believe such memory overhead is moderate and within an acceptable range for modern server and desktop systems, especially considering the current price drop of physical memory. In the meantime, we also recognize that such increase in memory consumption might be unwanted in certain memory-constrained applications, such as embedded systems. Because of that, our patch is provided as a compile-time option that may be enabled or disabled as needed.

6. DISCUSSION

As with many other real-world protection systems, our approach comes with a few limitations. First of all, for the purpose of verifying Linux kernel $W \oplus X$ enforcement, our approach assumes that the static kernel image and LKMs are trusted. More specifically, the kernel (including LKMs) is assumed to follow the transition rules (Section 2.2) to manage the kernel memory. If this assumption is violated, the invariants derived in this work may not be valid. Note that such trust can be potentially established by means of kernel/driver signing [12, 27, 28, 30], which falls outside the scope of this work. Second, it is important to note that while providing $W \oplus X$ is helpful to block code injection attacks, $W \oplus X$ itself does not prevent other types of attacks, such as "return-into-libc" [14]. Third, since our modeling is based on the correctness of the internal kernel APIs, any code inside the function that modifies page tables directly will not be prevented from doing so. Finally, although we establish a $W \oplus X$ property in the Linux kernel, there are a few exceptions that we had to consider for the implementation of our patches. For example, a special accommodation had to be made for kernel's built-in function tracing facility - *ftrace* [1]. The function tracing facility essentially requires dynamically modifying kernel code, which is in conflict with our $W \oplus X$ enforcement. As a result, we have to allow a short window of $W \oplus X$ violation while ftrace is in use to place its trace points (that requires modifying kernel's code at run time).

7. RELATED WORK

Model checking for improved security The first category of related work includes recent efforts that leverage model checking to improve systems security. For example, Mitchell et al. [22] applied model checking to successfully verify the correctness of (and find bugs in) security protocol specifications. Chen et al. [8] utilized model checking to analyze or demystify the confusing setuid system calls. Others [6, 7, 11, 29, 34] have used software model checking and static analysis to find a general class of bugs in source code. In contrast, our focus is on the analysis and verification of $W \oplus X$ enforcement for Linux kernel memory protection.

$W \oplus X$ enforcement The second category of related work aims at enforcing $W \oplus X$ as an effective defense against code injection attacks. For example, SecVisor [30] and NICKLE [28] use custom hypervisors to enforce code protection and data non-execution. Such protection is achieved through effectively separating code and data address spaces. Both methods make it impossible to modify code and to execute data without going through the special authentication mechanisms controlled by a hypervisor. Note that even with hardware-based full virtualization support, they inevitably lead to significant performance degradation, as policy enforcement requires additional management activities that consume extra clock cycles and cause cache pollution. Others take advantage of standard hardware protection features in the most straight-forward manner and introduce minimal impact. For examples, both PaX [25] and ExecShield [33] make use of standard memory protection features found in most architectures to achieve $W \oplus X$ property. Shared with our approach, these patches separate memory pages into two categories: code and data. Code pages are set as $RO + X$, while data as $RW + NX$. However, our approach is different from them in that we use a model checking approach to systematically analyze the $W \oplus X$ protection in the Linux kernel memory space while others mainly concentrate on userspace application protection. Also, our solution is based on the model-checking approach, which is appropriate for formal verification.

Other code injection defense mechanisms The third category of related work contains other approaches to defend against code injection attacks. For example, two other notable ways in this category include Address Space Layout Randomization (ASLR) [24, 25] and Instruction Set Randomization [15, 3]. ASLR is based on the idea of randomization of all major components within the application address space. This typically involves introducing random offsets in the layout of all major sections of the primary executable and the libraries it requires at link-time. This type of protection is already included in the mainline kernel and used exclusively for userspace. Instruction set randomization takes a somewhat different approach by randomizing the actual machine instruction set. This is achieved by creating a virtual machine with unique instruction encoding for each run. Compatibility with pre-compiled binaries is established by load-time binary translator, which converts the code from well-known "generic" instruction encoding to the encoding used in the specific virtual machine. Such approaches are not widely deployable since dynamic instruction sets are not supported by any modern hardware and software-based emulation likely introduces prohibitive performance overhead.

8. CONCLUSION

In this paper, we have presented a model checking-based approach to analyze the $W \oplus X$ protection in the Linux kernel space. Our modeling has led to the discovery of several real problems in the current Linux kernel design and implementation. Based on the model checking results, we have accordingly developed five kernel patches to fix them and four of them are in the process of being integrated into the mainline Linux kernel. Our evaluation with these patches indicate that they involve minimal changes and incur negligible performance overhead to the Linux kernel.

Acknowledgments The authors would like to thank the anonymous reviewers for their numerous, insightful comments that greatly helped improve the presentation of this paper. This work was supported in part by the US Army Research Office (ARO) under grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI), the US Air Force Research Laboratory (AFRL) under contract FA8750-09-1-0224, and the US National Science Foundation (NSF) under Grants 0852131, 0855297, 0855036, 0910767, and 0952640. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the ARO, the AFRL, and the NSF.

9. REFERENCES

[1] A Look at ftrace. `http://lwn.net/Articles/322666/`.

[2] Advanced Micro Devices. *AMD64 Architecture Programmerąŕs Manual Volume 2: System Programming*, 3.14 edition, September 2007.

[3] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *CCS '03: Proceedings of the 10th ACM Computer and Communications Security Conference*, 2003.

[4] T. C. Block. *Standard BIOS 32-bit Service Directory Proposal*. Phoenix Technologies Ltd., 0.4 edition, June 1993.

[5] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, third edition, 2005.

[6] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI'08: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2008.

[7] H. Chen and D. Wagner. MOPS: An Infrastructure for Examining Security Properties of Software. In *CCS'02: Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002.

[8] H. Chen, D. Wagner, and D. Dean. Setuid Demystified. In *Security '02: Proceedings of the 11th Conference on USENIX Security Symposium*, 2002.

[9] I. Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 3A: System Programming Guide, Part 1*. Intel Corp., 2006. Publication number 253668.

[10] D. L. Dill. The Murphi Verification System. `http://eprints.kfupm.edu.sa/70602/1/70602.pdf`, 1996.

[11] J. Franklin, A. Seshadri, N. Qu, S. Chaki, and A. Datta. Attacking, Repairing, and Verifying SecVisor: A Retrospective on the Security of a Hypervisor Research Area. Technical report, June 2008.

[12] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *SOSP '03: Proceedings of the 19th Symposium on Operating System Principles*, October 2003.

[13] M. Gorman. *Understanding The Linux Virtual Memory Manager*. Prentice Hall, May 2004.

[14] R. Hund, T. Holz, and F. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Security '09: Proceedings of the 18th Conference on USENIX Security Symposium*. USENIX Association, 2009.

[15] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *CCS '03: Proceedings of the 10th ACM Computer and Communications Security Conference*, 2003.

[16] The Linux Kernel Archives. `http://www.kernel.org`.

[17] S. Liakh and X. Jiang. [1/4,tip:x86/mm] correcting improper large page preservation. https://patchwork.kernel.org/patch/90045/, 2010.

[18] S. Liakh and X. Jiang. [2/4,tip:x86/mm] set first mb as rw+nx. https://patchwork.kernel.org/patch/90048/, 2010.

[19] S. Liakh and X. Jiang. [3/4,tip:x86/mm] nx protection for kernel data. https://patchwork.kernel.org/patch/90046/, 2010.

[20] S. Liakh and X. Jiang. [4/4,tip:x86/mm] ro/nx protection for loadable kernel modules. https://patchwork.kernel.org/patch/90047/, 2010.

[21] E. P. M. Michalis Polychronakis, Kostas G. Anagnostakis. An Empirical Study of Real-world Polymorphic Code Injection Attacks. In *LEET '09: Proceedings of the 2nd USENIX Workshop on*

Large-Scale Exploits and Emergent Threats. Usenix Association, April 2009.

[22] J. Mitchell, V. Shmatikov, and U. Stern. Finite-State Analysis of SSL 3.0. In *Security '98: Proceedings of the 7th Conference on USENIX Security Symposium*, 1998.

[23] A. Nayani, M. Gorman, and R. S. de Castro. Memory Management in Linux - Desktop Companion to the Linux Source Code. `http://www.ecsl.cs.sunysb.edu/elibrary/linux/mm/mm.pdf`, May 2002.

[24] PaX ASLR. `http://pax.grsecurity.net/docs/aslr.txt`.

[25] PaX NOEXEC. `http://pax.grsecurity.net/docs/noexec.txt`.

[26] PCI Special Interest Group. *PCI BIOS Specification*, 2.1 edition, August 1994.

[27] S. Pearson. Trusted Computing Platforms, the Next Security Solution. Technical report, HP Laboratories, November 2002.

[28] R. Riley, X. Jiang, and D. Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*. Springer-Verlag, 2008.

[29] B. Schwarz, H. Chen, D. Wagner, G. Morrison, J. West, J. Lin, and W. Tu. Model Checking An Entire Linux Distribution for Security Violations. In *ACSAC'05: Proceedings of the 21st Annual Computer Security Applications Conference*, 2005.

[30] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *SOSP '07: Proceedings of the 21st ACM SIGOPS symposium on Operating systems principles*. ACM, 2007.

[31] J. Tombs, B. Smith, R. Grehan, T. Yager, D. C. Niemi, and I. Smith. Unixbench-5.1.2. `http://code.google.com/p/byte-unixbench/`.

[32] Ubuntu. `http://www.ubuntu.com/`.

[33] A. van de Ven. Limiting Buffer Overflows with ExecShield. *Red Hat Magazine*, July 2005.

[34] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *OSDI'04: Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, 2004.

Appendix A: Defining the Initial State in the Model

```
procedure map_linear();
begin
 -- page count starts from 1, therefore +1
 for va: (page_offset+1) ..
       (page_offset+kernel_direct_size) do
    -- I1: linear kernel mapping
    pt[va].addr := va - page_offset;
    pt[va].present := true;
 end;
end;

procedure map_static_kernel();
begin
 -- page count starts from 1, therefore +1
 for va: (page_offset+1) ..
       (page_offset+kernel_direct_size) do

    -- I2: are we mapping code and ro-data?
    if(va <= (page_offset+kernel_text_size
              + kernel_ro_size))
    then
      -- yes, set page as R/O
      pt[va].prot.w := false;
    else
      -- no, set it as writable
      pt[va].prot.w := true;
    end;

    -- I2: are we mapping kernel data?
    if(va>(page_offset+kernel_text_size))
    then
  -- yes, set it non-executable
  pt[va].prot.x := false;
    else
      -- no, set code as executable
      pt[va].prot.x := true;
    end;
  end;
end;

procedure map_bios();
begin
 -- page count starts from 1, therefore +1
 for va: bios_start .. bios_end do
    -- I3: BIOS mapping
    pt[va].addr := va - page_offset;
    pt[va].present := true;
    pt[va].prot.w := true;
    pt[va].prot.x := true;
 end;
end;

startstate
begin
        clear pt;
        map_linear();
        map_static_kernel();
        map_bios();
end;
```

Back to Berferd

William Cheswick
AT&T Labs - Research
ches@research.att.com

ABSTRACT

It has been nearly twenty years since I published the Berferd paper. Much of it is quite outdated, reflecting the state of technology at the time. But it did touch a number of issues that have become quite important. I discuss some of the existing conditions around the time of the paper, and some of these issues.

Categories and Subject Descriptors

X [**Security**]: Internet history

1. INTRODUCTION

In 1500 AD, if you had a ship with enough food and a stout-hearted crew and you sailed west long enough, you were likely to discover some place unknown to the Europeans. In the late 1980s, if you had an Arpanet connection, some spare time, and a concern about security, you were likely to be working on something new and eventually, quite important.

There were some notable ships in the metaphorical westbound fleet. Digital Equipment had a substantial fleet. Notable sailors included Brian Reid, Jeff Mogul, Fred Avolio, and Marcus Ranum. I crewed for Dave Presotto at Bell Labs, and joined Steve Bellovin in a number of efforts. There was a dark, unmarked ship, probably sailing for the NSA, ahead of us in the distance. We didn't hear much from them in those days.

Some of these sailors, including me, have been dubbed "the father of the firewall" by the media. Most of these people, and a number of others, could plausibly have some claim to the title. But the world was ready for firewalls.

This paper looks back at my second Internet paper, *An Evening With Berferd, in Which a Hacker is Lured, Endured, and Studied*[7]. Much of the paper is hopelessly outdated—in Internet time, it dates from the pre-Cambrian era. I won't include the paper here nor discuss it in much detail. It is available on the Internet and in several places.[8, 9, 12, 7] (I particularly like the title *Une Soirée de Berferd* in the French translation of our Firewalls book.[4] But a number of themes I mentioned or assumed in the paper have become quite important. It is a snapshot of the early stages of ongoing arms races, which I will discuss below.

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

2. THE BERFERD MILIEU

The previous twenty years had seen an extensive amount of work in computer security. Multics[21] had been a platform for security ideas for years. Most of the security ideas taught today come from seminal works like Saltzer and Schroeder[26]. This is the deep magic of computer and network security.

The Orange Book[13] laid out security rules for isolating users at different security levels on a machine. Computing resources were expensive, and this document suggested techniques for sharing these resources while minimizing data leakage. This was a fading goal given the decreasing price of computing. Even in 1977 I saw expensive computers rebooted with different disk packs to run jobs at different security levels. The security was simply easier to implement and audit, which is always a good sign.

One main security concern addressed by the Orange Book was the prevention of leakage from high classification levels to low levels. It did not deal directly with the threat of importing viruses into high-level systems, flow in the opposite direction. Still, the Orange Book had a lot of useful security advice that is applicable today.

A lot of security lessons had been taught, if not learned, by the late 1980s. Much of this early work is overlooked by students today. Students wishing to catch up can find some good lists, *c.f.* references [23, 22].

Hardware design in the previous twenty years had experienced a Cambrian explosion[19] of its own, exploring a number of interesting solutions. IBM had virtual machines on its mainframes in the 1960s. Burroughs had computers whose security relied on the compilers—users were not allowed to write machine code themselves. The hardware checked array bounds and other limits. Code and data were separated, and data could not be executed. Seymore Cray built RISC machines from the start, though the name came later.

In the late 1980s the Internet was a zero billion dollar business. We were marveling that the technology worked, never mind the obvious security problems. Useful crypto was suppressed and essentially unavailable, though only export from the U.S. was controlled.

These were the last days of the distributed file system ("doofus") wars: there were a number of ways to share file systems. Alas, NFS won. Similar to the QWERTY key-

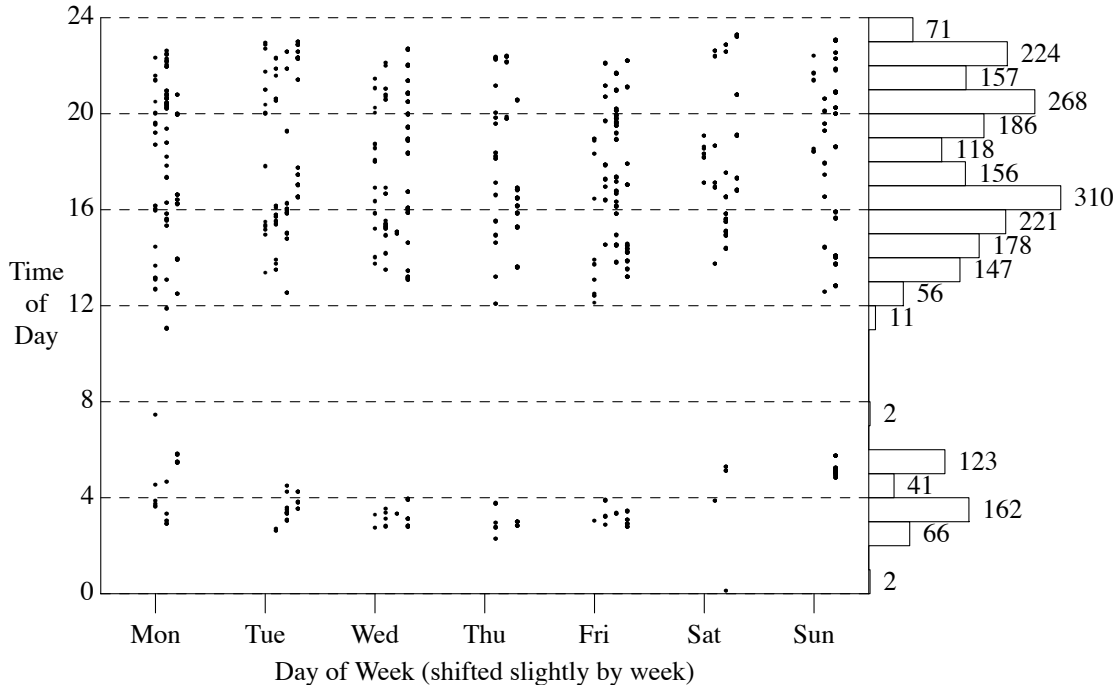


Figure 1: Here are the access times of the latter-day berferd who stole my iPhone. It is pretty clear he goes to sleep around 4AM (later on the weekends) and has a part-time job starting late in the evening. In the case of Berferd, we hoped we might divine his approximate time zone.

board as an early victor, it had important security flaws and did not correctly implement some Unix file system semantics. These problems have only been recently addressed in NFS version 4.

X11 was new and spreading. *Telnet*, *FTP*, and the Unix "r" commands (*rsh* and *rlogin*) were popular and useful. Unix was common (it was nearly 600,000,000 seconds old when the Morris Worm hit). DEC VMS and a number of other operating systems were common. Microsoft was not a factor, except perhaps in virus research—Windows 95 was years in the future. In fact, for many systems one had to purchase TCP/IP separately from the operating system; Woolongong was probably the most important supplier. Cisco was just emerging from the crowd as the leading router manufacturer.

There were a number of networks and related protocols, like Bitnet, CSnet, ACSnet, Usenet, *uucp*, MILnet, and Arpanet with TCP/IP. The latter was beginning to pull into the lead by the mid- to late-1980s. In Europe the carefully-engineered OSI stack (the "ISO tower") made inroads with X.25, but we only seem to need about five explicit layers in our network stack, not seven.

Many of these networks had their own email address formats, and I spent a fair amount of time in my early career at Bell Labs keeping the address translations straight. For example, the address `rdk%templevm.bitnet@cunyvm.cuny.edu` was correct, but we allowed `bitnet!templevm!rdk`.

I don't think that many of us realized at the time that short hallway conversations and quick technical suggestions would often each foreshadow billion-dollar businesses. Many ideas were discussed at the time, if not implemented or even patented. I declined to make a few patent applications because I misunderstood the legal meaning of "obvious." Most of us westward-bound sailors knew this stuff.

Of course, new names have emerged for some of these old ideas. We were doing plenty of *deep packet inspection* back then. Similarly, programmers had been dealing with code *refactoring* and *technical debt* since programs were first written.

We were native Internetians before the Great Domain Name Land Rush. I still have a little twinge when I see a new domain name on a commercial. "furniturewarehouses.com"! Darn! I could have had that! And I have friends who have made small fortunes because they picked good domain names. I've been offered money for cheswick.com: it's too bad I wasn't named MacDonalds.

Computer viruses were in the news. These spread in Microsoft hosts, usually via floppy disk. Most of them damaged the infected machine or displayed political messages. Fred Cohen had studied viruses a few years before.[10] At Bell Labs, Doug McIlroy had studied them, and Tom Duff was making management nervous with a virus spread in shell scripts.[14] The *probe.com* worm inadvertently took down Digital Equipment's corporate network for a couple of days.

Attacks were personal, with only the scans automated. Many scans were done by hand. The earliest vulnerability scanner I can remember was a set of shell scripts by Mike Muuse called *sweep*. This detected five different weaknesses in remotely network servers.

3. THE BELL LABS FIREWALL

The Morris Worm hit in November, 1988. I had been running Presotto's application gateway for almost a year. When I heard of the Worm's attack, I had a sinking feeling

that perhaps the Worm might have discovered a weakness in the firewall. I went into work.

At the Labs, all seemed safe. I remember Peter Weinberger calling one site and bragging that we hadn't been hit. The gateway had held up, but we had an unprotected connection to Bellcore, who was totally infested. The Morris Worm chose targets by running through `/etc/hosts` (a list of all computers known to the machine!) Our computers were at the end of their list, but the exponential growth of the worm was insufficiently controlled[15]. The Bellcore machines were so swamped, the worm never reached our entry at the end of the file. The worm would have spread quickly on the Bell Labs intranet: I found 1,300 hosts that were susceptible to it.

The worm was the first Internet Denial of Service (DOS) attack, though it wasn't meant to be. Controlling potentially exponential growth continues to be a problem for virus and worm writers today, though only when stealth is desired.

I did not like the uncomfortable feeling about our gateway and sought a simpler, more reliable design. This desire to implement security in a much more reliable way has guided my security choices ever since. Confidence in security should come from auditable simplicity.

I redesigned the Bell Labs gateway and described it in a paper.[6] This new gateway was festooned with logging: anything that wasn't explicitly understood was logged as suspicious. Steve Bellovin and I wanted to see what the Bad Guys were up to, motivated in part by Stoll's article[29]. We had a stronger defensive position than he had. Steve wrote a couple papers about the attacks on the firewall[2] and the odd packets we detected on the external network[3].

For a couple of years we chased down suspicious probes and reported them to the folks who ran the offending sites. The messages were of the sort: "We don't care very much, but we received the following network traffic that seems to indicate that there are hackers on machine X." About half of these emails got responses. I stopped do this after a while. It was like counting bugs on the windshield.

The sensitivity to network probes has mutated over the years. While I was sending out these letters, Fred Cohen was sending nasty messages to ISPs when unsolicited ping packets arrived on his network. By 2000, the Internet Mapping Project[5] would receive a couple complaints a month about our repeated traceroutes of networks. Five years later, the complaints were gone. The Internet now has so much 'background radiation" of packets that few note or care about some casual traffic. I recently measured about six packets per hour per IP address of unsolicited packets on unused IP addresses.

In the late 1980s, Mark Horton asked for a class A network for AT&T, and was given network 12.0.0.0/8. This network sat unused for a year or two: our internal routers couldn't handle the multiple subnetting such an address required. I announced the unused network to the Internet, directing any traffic to an unused MAC address, and using *tcpdump* to record the traffic. Since there were no hosts on the network, there should be no traffic. Of course, there was—up to 25MB per day. Much of this was backscatter from attacks on systems that used address-based authentication. This was the first packet telescope that I can recall, and certainly one of the largest. Of course, AT&T later found ample use for this network.

4. GLANCING THROUGH THE BERFERD PAPER

The paper is quite dated. Specific attacks and whole technologies have evolved and matured. The Internet gateway was hand-built out of various parts, as all firewalls were at the time. The design was solid, and never failed in unexpected ways throughout its lifetime.

The first half of the paper deals with attempted attacks and simulated responses to those attacks. I think the paper doesn't make clear that Berferd was attacking the firewall machine itself, drawing chalk figures on a case-hardened steel wall. There were fake services and a lot of logging to detect external interest. These were modified to make it look like the attack was working, eventually. With a human in the loop, the responses should have been very unconvincing. Steve Bellovin was watching the network traffic from a monitoring machine. We wanted that machine to be stealthy, so Steve cut the transmit wire to the Ethernet connection.

The attack on our gateway was minor: Berferd was causing lost sleep in many places, especially the Netherlands and at Stanford University. The timing of the attack on our gateway seemed oddly coincidental with the start of the first Gulf war. It raised the excitement at our end, but Berferd had been a problem long before that. Saddam was not using cyber warfare, though the thought crossed my mind at the time.

The second half of the paper describes a *jail* we set up. (To my knowledge this was the first use of the term in this context.) This was a separate machine, not the firewall itself, configured as what we now call a high-interaction honeypot. The implementation was highly idiosyncratic, using Datakit[16] to splice incoming connections through a separate logging machine. (Datakit was the predecessor to ATM and MPLS, and was widely used in AT&T in those days.) The host itself ran MIPS Unix, with some modifications, and an attractive file system to attract interest, similar to Cliff Stoll's SDInet. Berferd actually logged into a *chroot* environment within the target machine.

The traceback of the attacks was fairly successful, but it helped that the attackers had been watched from the attacking end.

The paper also includes an analysis of attack times. If the attacks are not automated, we can look at the time-of-day information and try to guess the attacker's time zone, sleep, and attack habits. We didn't learn much in Berferd's case.

(But this was useful recently when I tracked down my stolen iPhone. When the iPhone was active, it would attempt to log into my (disabled) mail server, leaving a trace of IP addresses suitable for a subpoena. The times from the mail logs are shown in Figure 1. You can draw some conclusions about the thief's daily schedule.)

Finally, Berferd came to our jail and used it to attack the world for three days. We were collecting real-time information about a hacker and his techniques and targets. This kind of information was hard to come by at the time. (Now, law enforcement and honeypot owners can get this information in bulk quantities easily.)

Management became upset, and made us shut down the honeypot. Like many activities at the time, we were ahead of the law, but there are plenty of liability statutes that could be invoked against us if desired.

5. THEMES

5.1 Law Enforcement, and the Law

Law enforcement was not up to speed in the cyber world in the late 1980s. There were a couple of notable prosecutors, like Bill Cook, who were making news in the area, but it was all new. There were some laws in the U.S., notably the Electronic Communications Privacy Act, that provided some guidance. But there was little in the way of treaties—we could not touch Berferd in the Netherlands. In many cases we were playing it by ear, trying to Do The Right Thing, whatever that meant. Was it an act of war to ping Finland? Leviticus does not discuss these issues.

A few days after Berferd's attacks through the honeypot ended, I had a visit from a couple of investigators from the U.S. Army Intelligence. Berferd had attacked some 300 computers while we watched, and I had been busy informing the targets of the attacks and weaknesses found. A number of the sites were U.S. military computers. (Of course, U.S. military computers have been under attack since about five minutes after they were connected to a network containing .edu machines. the military has had a lot of practice over the decades.)

I explained what had been going on, gave them lots of printouts, and they went away, mollified. I had been read my Miranda rights, and, in retrospect, really should have had a lawyer present.

Since then, there have been a number of opportunities to help train law enforcement and other government entities on advances in Internet security and forensics. I have been privileged to help train investigators, and even assisted in several cases. Sometimes, fifteen minutes of technical poking can help an investigation along. I had a small role at the start of the largest child pornography bust in history. I earned enough trust to be one of those who were allowed to assist the New York branch of the Secret Service in reconnecting to the network after 9/11.

These industry/government collaborations can add a great deal of flexibility to a national response when things go bad.

Law enforcement, at the state and especially the national level have been up to speed for quite a while. Their forensic labs are well-equipped and manned by sharp agents with post-graduate training. And every day they see the stuff that the Berferd project was designed to watch, but at a much more advanced level.

(The recovery of my iPhone mentioned above required the help of law enforcement. I got my phone back, and they learned some new forensic techniques. I am hoping to help make smart phones too dangerous to steal in the future.)

Of course, we have strong crypto now, easily available to those who bother to use it. It is a concern, and sometimes a problem, for law enforcement. But so far, they seem to have managed to deal with this: you don't go through security, you go around it. I think widely-available strong cryptography is a net win.

Cliff Stoll's investigations were started with a $0.75 accounting discrepancy, and a great deal of trouble emerged. For a while, law enforcement would judge the importance of attacks by the early estimates of losses from those attacks. But, as in Stoll's case, they often reveal a much larger pattern of trouble, and law enforcement sometimes tackles small cases now, having learned that they are often connected to other, larger cases.

It is hard to trace back connections on the Internet[27], but given enough time and motivation, it can be done. It does help to have the coöperation of the home country, and some previously unhelpful countries are finding that it is useful to be more coöperative.

5.2 VM/honeypot arms race

In the Berferd paper we implemented about the dumbest honeypot imaginable. It was human-powered, reacting at human speeds to *syslog* entries and notification emails. Berferd did not seem to notice, perhaps because we were leaving plausible explanations lying around. It certainly wouldn't work now.

Most attacks are automated now, at least at the start. This is especially true for targets of opportunity. They don't want to own your particular machine, they just want a collection of owned hosts. They have nothing personal against grandma, they just want to coöpt her computer a bit.

But targeted attacks are another thing entirely. Attackers are much more suspicious of traps and counter-espionage now. It is not that they are particularly likely to get caught: they are not, unless attacks persist for months. But they don't want to alert the target if possible. And they don't want to reveal their methods, either. Attack avenues are not infinite; exploits have a limited shelf life and can be used up.

Fred Cohen's *Deception Toolkit*[11] works to increase the attacker's uncertainty. Is the attacker making progress, or going deeper into a rat hole? In the Berferd incident, I wanted to waste the attacker's time. He won—I've spent a lot more time on this than he did—but delaying and confusing the enemy is a useful goal.

This arms race continues today, both in honey pots and virtual machines. The high- and low-interaction honeypots, and virtual machines, can be suitable for trapping and delaying attacks, and analyzing attack technologies. I suspect that in neither case will the defenders win this battle.

Simulations and honeypots can be detected, and virus writers have waged a long battle against the anti-virus community. Some early viruses could detect when they were being executed in a debugger, and behave differently. This battle continues between the increasing quality of the virtual machine engines, and detecting differences between a VM and a real host. Viruses are getting harder to detect, and ultimately the defenders are trying to solve the halting problem, which dooms them to failure. I think that restricting execution to signed code is more promising, but has its problems. And it is unlikely that timing and other idiosyncrasies will be completely eliminated from virtual machines.

5.3 IP packets are dangerous

At the Labs, and later in the Firewalls book, we maintained that IP packets were dangerous. They have numerous options and varying implementation details and, if you are interested in security, you should avoid letting them through your firewall. Instead, we recommended application- and circuit-level gateways.

Over the years, firewalls have switched to IP packet transmission for convenience and efficiency. It has worked well enough, but our warnings have been justified. *Firewalking,*[18] sending unusual packets through firewalls to probe the networks they guard, has been a fairly effective surveillance tool. It relies in part on weird IP packets. This is an ongoing threat that allows attacks on endpoints that intru-

sion detection systems have difficulty analyzing.

In fact, *packet normalizers*[20] have been used to create standard packet streams that resist idiosyncratic packet attacks and hide host implementation details. A circuit-level gateway seems safer.

The U.S. military networks have *guards* between networks of differing security levels. It is my understanding that they do not permit raw IP connectivity between these networks, though I would bet next week's salary that there are some. In any case, this makes the guards application-level gateways, (but we are not allowed to call them that, probably for political reasons.) I believe that these guards predate firewalls. (Come to think of it, the NSA and others in the military had a few fathers of the firewall as well.)

We have been criticized for advocating this IP-free approach to connectivity. Having a "computer acting as a wire" can add a point of failure, a performance bottleneck, and violates the end-to-end principle.[25] I agree with the value of end-to-end connectivity for innovation, but the pragmatics of weak host security still trump these concerns. Hosts *should* have strong host security, and my personal computers have been "skinny dipping" on the Internet since the mid-1990s, reaping benefits of clean end-to-end connectivity.

5.4 Viruses, and state actors

Computer viruses and worms are far more important now. They changed from instruments of destruction and carriers of propaganda to commercial entities for the underground economy. Rather than destroy hosts, these programs tend to repair security problems in the hosts they inhabit to keep other attacks out. This is another arms race, this time between malefactors.

Modern viruses co-opt computers for financial gain. A random PC is a fine machine for hosting a phishing web server, and grandma will never notice the difference. Profits go to organized crime and terrorists. Zombie armies of co-opted hosts are used in distributed denial of service attacks.

This technology is also useful for national actors: cyber warfare has become extremely competent. The recent "Aurora" attacks on Google, and especially the Stuxnet worm (*c.f.* Schneier[28] and Friedman[17]) show immense competence and patience typical of a well-organized, well-funded effort.

And anyone can play in national attacks. In May 1999, I used software from the Internet Mapping Project to watch the Serbian networks. I quickly located a web server with Serbian propaganda on it, included photos of dead babies. I could have taken this machine down: Suddenly the Republic of Cheswick needed a foreign policy, perhaps to the annoyance of those responsible for our actual national foreign policy.

The Internet brings world-changing technology and access into the hands of anyone with a computer and some spare time. Partisan amateurs attack the obvious targets all the time, and especially at times of heightened tension.

I do not recall any speculations of these kinds of attacks in the 1980s. But we were certainly thinking of the possibilities by the mid-1990s. I have anecdotal evidence that the U.S. had an offensive cyber capability in place by 2000.

6. CONCLUSION

Computer and network security is a big deal now. Our economies depend on these technologies, and we don't handle the tradeoff between security and convenience very well. Security and convenience don't have to be mutually exclusive (consider modern hotel key cards), but we have to get the engineering right.

There is no better example than the current situation with passwords. Sites require high-entropy passwords, but limit the number of login attempts, which defeats the dictionary attacks that strong passwords were designed to resist. But the users are still stuck with baroque eye-of-newt password rules. The strong passwords reduce security and convenience.

The mad rush to new software capabilities and product features prevents us from settling down and hardening tried-and-true technologies. For example, back in the Berferd era, the *sendmail* program was a continuing source of insecurity. The program was large and over-privileged, and trusted to transport most of the world's email. In fact, Dave Presotto wrote the *upas* mailer[24] to provide a much safer, simpler alternative on that first firewall.

We don't hear much about *sendmail* problems any more. This is partly because there are better targets on Unix hosts (*i.e.* PHP).[1] But the program has been annealing in the Internet crucible for a long time, and it still transports a large amount of mail with safety. We actually do make progress when software is allowed to settle down.

A few years ago, I found a bug in the Unix *cp* command. (You can imagine how hard this was to find!) This was a command that probably should have been frozen some time during the Carter administration. But someone had added an "optimization" to read the source file using the kernel's memory-mapped file feature. This, and hence *cp*, was broken for the source file system I was using.

Of course, security problems will continue, though I retain hope that we may slowly get better at this. Perhaps as Windows 7 supplants Windows XP, client systems will actually be more robust to attacks and we will slowly starve the market for zombie volunteers.

I am certain that exploitation of the human element will continue to be fruitful, despite most of our efforts. Phishing is just the beginning.

Some have suggested that we need to re-engineer the Internet, or perhaps create a new one; and this time, using crypto, we will insist that all endpoints be identifiable. I don't see that such a new network would be able to avoid our current problems.

The problem of attribution is clear. In the Berferd case, Weitse Venema was close to the source and able to verify at least one of the attackers, but we were never completely sure who Berferd was. That kind of information is harder to get now.

If an attack comes from a country, are they responsible for it, regardless of the actual source? The Treaty of Westphalia says so. Under this, hot pursuit might be appropriate if the local country does not coöperate. And what about false flag operations? Human intelligence (HUMINT) will continue to be an important source of this information.

7. ACKNOWLEDGMENTS

Steve Bellovin provided a number of ideas, and refreshed my memory on a lot of this ancient history. Dave Kormann had helpful suggestions for this paper. Peter Neumann provided some pointers to the Deep Magic of computer security.

8. REFERENCES

[1] the month of php security. http://php-security.org/index.html, 2010.

[2] Steven M. Bellovin. There be dragons. In *Proceedings of the Third Usenix Unix Security Symposium*, pages 1–16, September 1992.

[3] Steven M. Bellovin. Packets found on an internet. *Computer Communications Review*, 23(3):26–31, July 1993.

[4] Bill Cheswick and Steve Bellovin. *Firewalls et sécurité Internet*. Addison-Wesley France, 1996.

[5] Bill Cheswick, Hal Burch, and Steve Branigan. Mapping and visualizing the internet. In *Usenix*, 2000.

[6] William R. Cheswick. The design of a secure internet gateway. In *Proc. Summer USENIX Conference*, Anaheim, CA, June 1990.

[7] William R. Cheswick. An evening with Berferd, in which a cracker is lured, endured, and studied. In *Proc. Winter USENIX Conference*, pages 163–174, San Francisco, CA, January 1992.

[8] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, Reading, MA, first edition, 1994.

[9] William R. Cheswick, Steven M. Bellovin, and Aviel D. Rubin. *Firewalls and Internet Security; Repelling the Wily Hacker*. Addison-Wesley, Reading, MA, second edition, 2003.

[10] F. Cohen. Computer viruses: theory and experiments. *Comput. Secur.*, 6(1):22–35, 1987.

[11] Fred Cohen. Depection toolkit. http://all.net/, 1998.

[12] Dorothy E. Denning and Peter J. Denning. *Internet Besieged*. Addison Wesley Professional, 1997.

[13] DoD trusted computer system evaluation criteria. DoD 5200.28-STD, DoD Computer Security Center, 1985.

[14] Tom Duff. Experience with viruses on UNIX systems. *j-COMP-SYS*, 2(2):155–171, Spring 1989.

[15] M. W. Eichin and J. A. Rochlis. With microscope and tweezers: An analysis of the Internet virus of november 1988. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 326–345, Oakland, CA, May 1989.

[16] A. G. Fraser. Proc. icc. pages 20.1.1–20.1.3, June 1979.

[17] George Friedman. The stuxnet computer worm and the iranian nuclear program—stratfor. http://www.stratfor.com/analysis/20100924_stuxnet_computer_worm_and_iranian_nuclear_program, 2010.

[18] David Goldsmith and Michael Schiffman. Firewalking: A traceroute-like analysis of IP packet responses to determine gateway access control lists, 1998.

[19] Stephen J. Gould. *Wonderful Life: The Burgess Shale and the Nature of History*. W. W. Norton and Company, 1990.

[20] M. Handley, C. Kreibich, and V. Paxson. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. *Proceedings of the USENIX Security Symposium*, pages 115–131, 2001.

[21] Paul A. Karger and Roger R. Schell. Multics security evaluation: Vulnerability analysis, Volume II. Technical Report ESD-TR-74-193, HQ Electronic Systems Division: Hanscom AFB, MA, June 1974.

[22] Douglas Maughan. The need for a national cybersecurity research and development agenda. *Commun. ACM*, 53(2):29–31, 2010.

[23] Peter G. Neumann, Matt Bishop, Sean Peisert, and Marv Schaefer. Reflections on the 30th Anniversary of the IEEE Symposium on Security and Privacy. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, Oakland/Berkeley, CA, May 16–19, 2010.

[24] David L. Presotto. *Upas*—a simpler approach to network mail. In *USENIX Conference Proceedings*, pages 533–538, Portland, OR, Summer 1985.

[25] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, 1984.

[26] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems, 1975.

[27] Stefan Savage, David Wetherall, Anna Karlin, and Tom Anderson. Practical network support for ip traceback. pages 295–306, 2000.

[28] Bruce Schneier. The stuxnet worm. http://www.schneier.com/blog/archives/2010/09/the_stuxnet_wor.html, 2010.

[29] Cliff Stoll. Stalking the wily hacker. *Communications of the ACM*, 31(5):484, May 1988.

Comprehensive Shellcode Detection using Runtime Heuristics

Michalis Polychronakis
Columbia University, USA
mikepo@cs.columbia.edu

Kostas G. Anagnostakis
Niometrics, Singapore
kostas@niometrics.com

Evangelos P. Markatos
FORTH-ICS, Greece
markatos@ics.forth.gr

ABSTRACT

A promising method for the detection of previously unknown code injection attacks is the identification of the shellcode that is part of the attack vector using payload execution. Existing systems based on this approach rely on the self-decrypting behavior of polymorphic code and can identify only that particular class of shellcode. Plain, and more importantly, *metamorphic* shellcode do not carry a decryption routine nor exhibit any self-modifications and thus both evade existing detection systems. In this paper, we present a comprehensive shellcode detection technique that uses a set of runtime heuristics to identify the presence of shellcode in arbitrary data streams. We have identified fundamental machine-level operations that are inescapably performed by different shellcode types, based on which we have designed heuristics that enable the detection of plain and metamorphic shellcode regardless of the use of self-decryption. We have implemented our technique in Gene, a code injection attack detection system based on passive network monitoring. Our experimental evaluation and real-world deployment show that Gene can effectively detect a large and diverse set of shellcode samples that are currently missed by existing detectors, while so far it has not generated any false positives.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Invasive software

General Terms

Security

Keywords

Shellcode Detection, Payload Execution, Code Emulation

1. INTRODUCTION

Code injection attacks have become one of the primary methods of malware spreading. In a typical code injection attack, the attacker sends a malicious input that exploits a memory corruption vulnerability in a program running on the victim's computer. The injected code, known as *shellcode*, carries out the first stage of the attack, which usually involves the download and execution of a malware binary on the compromised host.

Once sophisticated tricks of the most skilled virus authors, advanced evasion techniques like code obfuscation and polymorphism are now the norm in most instances of malicious code [19]. The wide availability of ready-to-use shellcode construction and obfuscation toolkits and the discovery rate of new vulnerabilities have rendered exploit or vulnerability specific detection techniques ineffective [31]. A promising approach for the generic detection of code injection attacks is to focus on the identification of the shellcode that is indispensably part of the attack vector, a technique initially known as abstract payload execution [33]. Identifying the presence of the shellcode itself allows for the detection of previously unknown attacks without caring about the particular exploitation method used or the vulnerability being exploited.

Initial implementations of this approach attempt to identify the presence of shellcode in network inputs using static code analysis [33–35]. However, methods based on static analysis cannot effectively handle malicious code that employs advanced obfuscation tricks such as indirect jumps and self-modifications. Dynamic code analysis using emulation is not hindered by such obfuscations and can detect even extensively obfuscated shellcode. This kind of "actual" payload execution has proved quite effective in practice [22] and is being used in network-level and host-level systems for the zero-day detection of both server-side and client-side code injection attacks [9, 14, 15, 23, 38].

A limitation of the above techniques is that they are confined to the detection of a particular class of polymorphic shellcode that exhibits self-decrypting behavior. Although shellcode "packing" and encryption are commonly used for evading signature-based detectors, attackers can achieve the same or even higher level of evasiveness without the use of self-decrypting code, rendering above systems ineffective. Besides code encryption, polymorphism can instead be achieved by mutating the actual instructions of the shellcode before launching the attack—a technique known as *metamorphism* [32]. Metamorphism has been widely used by virus authors and thus can trivially be applied for shellcode mutation. Surprisingly, even *plain* shellcode, i.e., shellcode that does not change across different instances, is also not detected by existing payload execution methods. Technically, a plain shellcode is no different than any instance of metamorphic shellcode, since both do not carry a decryption routine nor exhibit any self-modifications or dynamic code generation. Consequently, an attack that uses a previously unknown static analysis-resistant plain shellcode will manage to evade existing detection systems.

In this paper, we present a comprehensive shellcode detection technique based on payload execution. In contrast to previous ap-

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

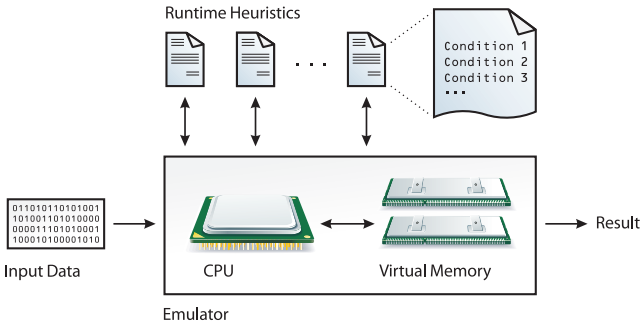


Figure 1: Overview of the proposed shellcode detection architecture.

proaches that use a single detection algorithm for a particular class of shellcode, our method relies on several runtime heuristics tailored to the identification of different shellcode types. We have designed four heuristics for the detection of plain and metamorphic shellcode targeting Windows systems. Polymorphic shellcode is in essence a self-decrypting version of a plain shellcode, and thus it is also effectively detected, since the concealed plain shellcode is revealed during execution. In fact, we also enable the detection of polymorphic shellcode that uses SEH-based GetPC code, which is currently not handled by existing polymorphic shellcode detectors. Furthermore, instead of solely using a CPU emulator, our approach couples the heuristics with an appropriate image of the complete address space of a real process, enabling the correct execution of shellcode that depends on certain kinds of host-level context.

We have implemented the above technique in Gene, a network-level detector that scans all client-initiated streams for code injection attacks against network services. Gene is based on passive network monitoring, which offers the benefits of easy large-scale deployment and protection of multiple hosts using a single sensor, while it allows us to test the effectiveness of our technique in real-world environments. Nevertheless, although Gene operates at the network level, its core inspection engine can analyze arbitrary data coming from any source. This allows our approach to be readily embedded in existing systems that employ emulation-based detection in other domains, e.g., for the detection of malicious websites [15] or in browser add-ons for the detection of drive-by download attacks [14].

Our evaluation with publicly available shellcode samples and shellcode construction toolkits, shows that Gene can effectively detect many different shellcode instances without prior knowledge about each particular implementation. At the same time, after extensive testing of the runtime heuristics using a large and diverse set of generated and real data, in addition to a five-month deployment in production networks, Gene has not generated any false positives.

2. ARCHITECTURE

The proposed shellcode detection system is built around a CPU emulator that executes valid instruction sequences found in the inspected input. An overview of our approach is illustrated in Fig. 1. Each input is mapped to an arbitrary location in the virtual address space of a supposed process, and a new execution begins from each and every byte of the input, since the position of the first instruction of the shellcode is unknown and can be easily obfuscated. The detection engine is based on multiple heuristics that match runtime patterns inherent in different types of shellcode. During execution, the system checks several conditions that should all be satisfied in order for a heuristic to match some shellcode. Moreover, new heuristics can easily be added due to the extensible nature of the system.

Abbreviation	Matching Shellcode Behavior
PEB	`kernel32.dll` base address resolution
BACKWD	`kernel32.dll` base address resolution
SEH	Memory scanning / SEH-based GetPC code
SYSCALL	Memory scanning

Table 1: Overview of the shellcode detection heuristics used in Gene.

Existing polymorphic shellcode detection methods focus on the identification of self-decrypting behavior, which can be simulated without any host-level information [23]. For example, accesses to addresses other than the memory area of the shellcode itself are ignored. However, shellcode is meant to be injected into a running process and it usually accesses certain parts of the process' address space, e.g., for retrieving and calling API functions. In contrast to previous approaches, the emulator used in our system is equipped with a fully blown virtual memory subsystem that handles all user-level memory accesses and enables the initialization of memory pages with arbitrary content. This allows us to populate the virtual address space of the supposed process with an image of the mapped pages of a process taken from a real system.

The purpose of this functionality is twofold: First, it enables the construction of heuristics that check for memory accesses to process-specific data structures. Although the heuristics presented in this paper target Windows shellcode, and thus the address space image used in conjunction with these heuristics is taken from a Windows process, some other heuristic can use a different memory image, e.g., taken from a Linux process. Second, this allows to some extent the correct execution of non-self-contained shellcode that may perform accesses to known memory locations for evasion purposes [10]. We discuss this issue further in Sec. 6.

3. RUNTIME HEURISTICS

Each heuristic used in Gene is composed of a sequence of conditions that should *all* be satisfied *in order* during the execution of malicious code. Table 1 gives an overview of the four heuristics presented in this section. The heuristics focus on the identification of the first actions of different shellcode types, according to their functionality, regardless of any self-decrypting behavior.

3.1 Resolving kernel32.dll

The typical end goal of the shellcode is to give the attacker full control of the victim system. This usually involves just a few simple operations, such as downloading and executing a malware binary on the compromised host. These operations require interaction with the OS through the system call interface, or in case of Microsoft Windows, through the user-level Windows API.

The Windows API is divided into several dynamic load libraries (DLLs). In order to call an API function, the shellcode must first find its absolute address in the address space of the process. This can be achieved in a reliable way by searching for the Relative Virtual Addresses (RVAs) of the function in the Export Directory Table (EDT) of the DLL. The absolute Virtual Memory Address (VMA) of the function can then be easily computed by adding the DLL's base address to the function's RVA. In fact, `kernel32.dll` provides the quite convenient functions `LoadLibrary`, which loads the specified DLL into the address space of the calling process and returns its base address, and `GetProcAddress`, which returns

```
1  xor eax, eax            ; eax = 0
2  mov eax, fs:[eax+0x30]  ; eax = PEB
3  mov eax, [eax+0x0C]     ; eax = PEB.LoaderData
4  mov esi, [eax+0x1C]     ; esi = InInitializationOrder
                                   ModuleList.Flink
5  lodsd                   ; eax = 2nd list entry
                                   (kernel32.dll)
6  mov eax, [eax+0x08]     ; eax = LDR_MODULE.BaseAddress
```

Figure 2: A typical example of code that resolves the base address of `kernel32.dll` through the PEB.

the address of an exported function from the specified DLL. After resolving these two functions, any other function in any DLL can be loaded and used directly. However, custom function searching using hashes is usually preferable in modern shellcode, since `GetProcAddress` takes as argument the actual name of the function to be resolved, which increases the shellcode size considerably.

No matter which method is used, a common fundamental operation in all above cases is that the shellcode has to first locate the base address of `kernel32.dll`. Since this is an inherent operation that must be performed by any Windows shellcode that needs to call a Windows API function, it is a perfect candidate for the development of a generic shellcode detection heuristic.

3.1.1 Process Environment Block

Probably the most reliable and widely used technique for determining the base address of `kernel32.dll` takes advantage of the Process Environment Block (PEB), a user-level structure that holds extensive process-specific information. Figure 2 shows a typical example of PEB-based code for resolving `kernel32.dll`. The shellcode first gets a pointer to the PEB (line 2) through the Thread Information Block (TIB), which is always accessible at a zero offset from the segment specified by the FS register. A pointer to the PEB exists 0x30 bytes into the TIB, as shown in Fig. 3. The absolute memory address of the TIB and the PEB varies among processes, and thus the only reliable way to get a handle to the PEB is through the FS register, and specifically, by reading the pointer located at address `FS:[0x30]`.

Condition P1. This fundamental constraint is the basis of our first detection heuristic **(PEB)**. If during the execution of some input the following condition is true **(P1)**: *(i) the linear address of `FS:[0x30]` is read, and (ii) the current or any previous instruction involved the FS register*, then this input may correspond to a shellcode that resolves `kernel32.dll` through the PEB.

The second predicate is necessary for two reasons. First, it is useful for excluding random instructions in benign inputs that happen to read from the linear address of `FS:[0x30]` without involving the FS register. For example, if `FS:[0x30]` corresponds to address `0x7FFDF030` (as shown in the example of Fig. 3), the following code will correctly not match the above condition:

```
mov ebx, 0x7FFD0000
mov eax, [ebx+0xF030]  ; eax = FS:[0x30]
```

On the other hand, the memory access to `FS:[0x30]` can be made through an instruction that does not use the FS register directly. For example, an attacker could take advantage of other segment registers and replace the first two lines in Fig. 2 with:

```
mov ax, fs             ; ax = fs
mov bx, es             ; preserve es
mov es, ax             ; es = fs
mov eax, es:[0x30]     ; load FS:[0x30] to eax
mov es, bx             ; restore es
```

The code loads the segment selector of the FS register to ES (`mov` between segment registers is not supported), reads the pointer to the PEB, and then restores the original value of the ES register.

The linear address of the TIB is also contained in the TIB itself at the location `FS:[0x18]`, as shown in Fig. 3. Thus, another way of reading the pointer to the PEB without using the FS register in the same instruction is the following:

```
xor eax,eax            ; eax = 0
xor eax,fs:[eax+0x18]  ; eax = TIB address
mov eax,[eax+0x30]     ; eax = PEB address
```

Note in the above example that other instructions besides `mov` can be used to indirectly read a memory address through the FS register (`xor` in this case). No matter how obfuscated the code is, the condition remains robust since it does not rely on the execution of particular instructions.

Although condition P1 is quite restrictive, the possibility of encountering a random read from `FS:[0x30]` during the execution of some benign input is not negligible. Thus, it is desirable to strengthen the heuristic with more operations exhibited by any PEB-based `kernel32.dll` resolution code.

Condition P2. Having a pointer to the PEB, the next step of the shellcode is to obtain a pointer to the `PEB_LDR_DATA` structure that holds the list of loaded modules (line 3 in Fig. 2). Such a pointer exists 0xC bytes into the PEB, in the `LoaderData` field. Since this is the only available reference to that data structure, the shellcode unavoidably has to read the `PEB.LoaderData` pointer. We can use this constraint as a second condition for the PEB heuristic **(P2)**: *the linear address of `PEB.LoaderData` is read*.

Condition P3. Moving on, the shellcode has to walk through the loaded modules list and locate the second entry (`kernel32.dll`). A pointer to the first entry of the list exists in the `InInitializationOrderModuleList.Flink` field located 0x1C bytes into the `PEB_LDR_DATA` structure. The read operation from this memory location (line 4 in Fig. 2) allows for strengthening further the detection heuristic with a third condition.

Although this is the most well known [5,26,27], and widely used technique for all Windows versions up to Windows Vista, it does not work "as-is" for Windows 7. In that version, `kernel32.dll` is found in the third instead of the second position in the modules list [7]. A more generic and robust technique is to walk through the list and check the actual name of each module until `kernel32.dll` is found [7, 29]. In fact, the `PEB_LDR_DATA` structure contains two more lists of the loaded modules that differ in the order of the DLLs. All three lists are implemented as doubly linked lists, and their corresponding `LIST_ENTRY` records contain two pointers to the first (`Flink`) and last (`Blink`) entry in the list.

Based on the above, and given that (i) `kernel32.dll` can be resolved through any of the three lists, and (ii) list traversing can be made in both directions, the third condition of the heuristic can be specified as follows **(P3)**: *the linear address of any of the `Flink` or `Blink` pointers in the `InLoadOrderModuleList`, `InMemoryOrderModuleList`, or `InInitializationOrderModuleList` records of the `PEB_LDR_DATA` structure is read*.

3.1.2 Backwards Searching

An alternative technique for locating `kernel32.dll` is to find a pointer that points somewhere into the memory area where the `kernel32.dll` has been loaded, and then search backwards until the beginning of the DLL is located [27]. A reliable way to obtain a pointer into the address space of `kernel32.dll` is to take advantage of the Structured Exception Handling (SEH) mechanism of Windows [21], which provides a unified way of handling hardware and software exceptions. When an exception occurs, the exception dispatcher walks through a list of exception handlers for

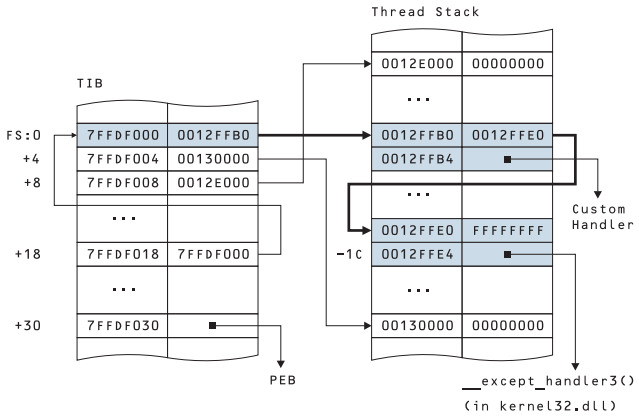


Figure 3: A snapshot of the TIB and the stack memory areas of a typical Windows process. The SEH chain consisting of two nodes is highlighted.

the current thread and gives each handler the opportunity to handle the exception or pass it on to the next handler. The list is stored on the stack of each thread, and each node is a SEH frame that consists of two pointers to the next frame and the actual handler routine. Figure 3 shows a typical snapshot of the TIB and the stack memory areas of a process with two SEH handlers. A pointer to the current SEH frame exists in the first field of the Thread Information Block and is always accessible through `FS:[0]`.

At the end of the SEH chain (bottom of the stack) there is a default exception handler that is registered by the system for every thread. The `Handler` pointer of this SEH record points to a routine that is located in `kernel32.dll`, as shown in Fig. 3. Thus, the shellcode can start from `FS:[0]` and walk the SEH chain until reaching the last SEH frame, and from there get a pointer into `kernel32.dll` by reading its `Handler` field.

Another technique to reach the last SEH frame, known as "TOPSTACK" [27], uses the stack of the exploited thread. The default exception handler is registered by the system during thread creation, making its relative location from the bottom of the stack fairly stable. Although the absolute address of the stack may vary, a pointer to the bottom of the stack is always found in the second field of the TIB at `FS:[0x4]`. The `Handler` pointer of the default SEH handler can then be found 0x1C bytes into the stack, as shown in Fig. 3. In fact, the TIB contains a second pointer to the top of the stack at `FS:[0x8]`.

Condition B1. Based on the same approach as in the previous section, the first condition for the detection heuristic **(BACKWD)** that matches the "backwards searching" method is the following **(B1)**: *(i) any of the linear address between `FS:[0]`–`FS:[0x8]` is read, and (ii) the current or any previous instruction involved the `FS` register*. The rationale is that a shellcode that uses the backwards searching technique should unavoidably read either i) the memory location at `FS:[0]` for walking the SEH chain, or ii) one of the locations at `FS:[0x4]` and `FS:[0x8]` for accessing the stack directly.

Condition B2. In any case, the code will reach the default exception record on the stack and read its `Handler` pointer. Since this is a mandatory operation for landing into `kernel32.dll`, we can use this dependency as our second condition **(B2)**: *the linear address of the `Handler` field of the default SEH handler is read*.

Condition B3. Finally, during the backwards searching phase, the shellcode will inevitably perform several memory accesses to the address space of `kernel32.dll` in order to check whether each 64KB-aligned address corresponds to the base address of the DLL. In our experiments with typical code injection attacks in Windows XP, the shellcode performed at least four memory reads in `kernel32.dll`. Thus, after the first two conditions have been met, we expect to encounter **(B3)**: *at least one memory read form the address space of `kernel32.dll`*.

3.2 Process Memory Scanning

Some memory corruption vulnerabilities allow only a limited space for the injected code—usually not enough for a fully functional shellcode. In most such exploits though the attacker can inject a second, much larger payload which however will land at a random, non-deterministic location, e.g., in a buffer allocated in the heap. The first-stage shellcode can then sweep the address space of the process and search for the second-stage shellcode (also known as the "egg"), which can be identified by a long-enough characteristic byte sequence. This type of first-stage payload is known as "egg-hunt" shellcode [28].

Blindly searching the memory of a process in a reliable way requires some method of determining whether a given memory page is mapped into the address space of the process. In the rest of this section, we describe two known memory scanning techniques and the corresponding detection heuristics that can capture these behaviors, and thus, identify the execution of egg-hunt shellcode.

3.2.1 SEH

The first memory scanning technique takes advantage of the structured exception handling mechanism and relies on installing a custom exception handler that is invoked in case of a memory access violation.

Condition S1. As discussed in Sec. 3.1.2, the list of SEH frames is stored on the stack, and the current SEH frame is always accessible through `FS:[0]`. The first-stage shellcode can register a custom exception handler that has priority over all previous handlers in two ways: create a new SEH frame and adjust the current SEH frame pointer of the TIB to point to it [28], or directly modify the `Handler` pointer of the current SEH frame to point to the attacker's handler routine. In the first case, the shellcode must update the SEH list head pointer at `FS:[0]`, while in the second case, it has to access the current SEH frame in order to modify its `Handler` field, which is only possible by reading the pointer at `FS:[0]`. Thus, the first condition of the SEH-based memory scanning detection heuristic **(SEH)** is **(S1)**: *(i) the linear address of `FS:[0]` is read or written, and (ii) the current or any previous instruction involved the `FS` register*.

Condition S2. Another mandatory operation that will be encountered during execution is that the `Handler` field of the custom SEH frame (irrespectively if its a new frame or an existing one) should be modified to point to the custom exception handler routine. This operation is reflected by the second condition **(S2)**: *the linear address of the `Handler` field in the custom SEH frame is or has been written*. Note that in case of a newly created SEH frame, the `Handler` pointer can be written before or after `FS:[0]` is modified.

Condition S3. Although the above conditions are quite constraining, we can apply a third condition by exploiting the fact that upon the registration of the custom SEH handler, the linked list of SEH frames should be valid. In the risk of stack corruption, the exception dispatcher routine performs thorough checks on the integrity of the SEH chain, e.g., ensuring that each SEH frame is dword-

```
1  push  edx       ; preserve edx across system call
2  push  0x8
3  pop   eax       ; eax = NtAddAtom
4  int   0x2e      ; system call
5  cmp   al, 0x05  ; check for STATUS_ACCESS_VIOLATION
6  pop   edx       ; restore edx
```

Figure 4: A typical system call invocation for checking if the supplied address is valid.

aligned within the stack and is located higher than the previous SEH frame [21]. Thus, the third condition requires that **(S3)**: *starting from* `FS:[0]`*, all SEH frames should reside on the stack, and the* `Handler` *field of the last frame should be set to 0xFFFFFFFF*. In essence, the above condition validates that the custom handler registration has been performed correctly.

3.2.2 System Call

The extensive abuse of the SEH mechanism in various memory corruption vulnerabilities led to the introduction of SafeSEH, a linker option that produces a table with all the legitimate exception handlers of the image. In case the exploitation of some SafeSEH-protected vulnerable application requires the use of egg-hunt shellcode, an alternative but less reliable method for safely scanning the process address space is to check whether a page is mapped—before actually accessing it—using a system call [27, 28]. As already discussed, although the use of system calls in Windows shellcode is not common, since they are prone to changes between OS versions and do not provide crucial functionality such as network access, they can prove useful for determining if a memory address is accessible.

Some Windows system calls accept as an argument a pointer to an input parameter. If the supplied pointer is invalid, the system call returns with a return value of `STATUS_ACCESS_VIOLATION`. Thus, the egg-hunt shellcode can check the return value of the system call, and proceed accordingly by searching for the egg or moving on to the next address [28]. In Windows, a system call is initiated by generating a software interrupt through the `int 0x2e` instruction.

Figure 4 shows a typical code that checks the address stored in `edx` using the `NtAddAtom` system call. In Windows, a system call is initiated by generating a software interrupt through the `int 0x2e` instruction (line 4). The actual system call that is going to be executed is specified by the value stored in the `eax` register (line 3). Upon return from the system call, the code checks if the return value equals the code for `STATUS_ACCESS_VIOLATION`. The actual value of this code is 0xC0000005, but checking only the lower byte is enough in return for more compact code (line 5).

Condition C1. System call execution has several constraints that can be used for deriving a detection heuristic for this kind of egg-hunt shellcode. First, the immediate operand of the `int` instruction should be set to 0x2E. Looking just for the `int 0x2e` instruction is clearly not enough since any two-byte instruction will be encountered roughly once every 64KB of arbitrary binary input. However, when encountering an `int 0x2e` instruction that corresponds to an actual system call execution, the `ebx` register should also have been previously set to the proper system call number.

The publicly available egg-hunt shellcode implementations we found (see Sec. 5.1) use one of the following system calls: `NtAccessCheckAndAuditAlarm` (0x2), `NtAddAtom` (0x8), and `NtDisplayString` (0x39 in Windows 2000, 0x43 in XP, 0x46 in 2003 Server, and 0x7F in Vista). The variability of the system call number for `NtDisplayString` across the different Windows versions is indicative of the complexity introduced in an exploit by the direct use of system calls. Based on the above, a necessary condition during the execution of a system call in egg-hunt shellcode is **(C1)**: *the execution of an* `int 0x2e` *instruction with the* `eax` *register set to one of the following values: 0x2, 0x8, 0x39, 0x43, 0x46, 0x7F*.

Condition C2. As shown in Sec. 5.2.2, condition C1 alone can happen to hold true during the execution of random code, although rarely. However, the heuristic can be strengthened based on the following observation. The egg-hunt shellcode will have to scan a large part of the address space until it finds the egg. Even when assuming that the egg can be located only at the beginning of a page [37], the shellcode will have to search hundreds or thousands of addresses, e.g., by repeatedly calling the code in Fig. 4 in a loop. Hence, condition C1 will hold several times. The detection heuristic **(SYSCALL)** can then be defined as a meta-condition **(C{N})**: *C1 holds true N times*. As shown in Sec. 5.2.2, a value of N = 2 does not produce any false positives.

In case other system calls can be used for validating an arbitrary address, they can easily be included in the above condition. Starting from Windows XP, system calls can also be made using the more efficient `sysenter` instruction if it is supported by the system's processor. The above heuristic can easily be extended to also support this type of system call invocation.

3.3 SEH-based GetPC Code

Before decrypting itself, polymorphic shellcode needs to first find the absolute address at which it resides in the address space of the vulnerable process. The most widely used types of GetPC code for this purpose rely on some instruction from the `call` or `fstenv` instruction groups [23]. These instructions push on the stack the address of the following instruction, which can then be used to calculate the absolute address of the encrypted code. However, this type of GetPC code cannot be used in purely alphanumeric shellcode [19], because the opcodes of the required instructions fall outside the range of allowed ASCII bytes. In such cases, the attacker can follow a different approach and take advantage of the SEH mechanism to get a handle to the absolute memory address of the injected shellcode [30].

When an exception occurs, the system generates an exception record that contains the necessary information for handling the exception, including a snapshot of the execution state of the thread, which contains the value of the program counter at the time the exception was triggered. This information is stored on the stack, so the shellcode can register a custom exception handler, trigger an exception, and then extract the absolute memory address of the faulting instruction. By writing the handler routine on the heap, this technique can work even in Windows XP SP3, bypassing any SEH protection mechanisms [30].

In essence, the SEH-based memory scanning detection heuristic described in Sec. 3.2.1 does not identify the scanning behavior per se, but the proper registration of a custom exception handler. Although this is an inherent operation of any SEH-based egg-hunt shellcode, any shellcode that installs a custom exception handler can be detected, including polymorphic shellcode that uses SEH-based GetPC code.

4. IMPLEMENTATION

We have implemented the proposed detection method in Gene, a network-level attack detector that uses a custom IA-32 emulator to identify the presence of shellcode in network streams. Gene scans the client-initiated part of each TCP connection using the runtime heuristics presented in this work. For evaluation purposes, a fifth

GetPC-based self-decrypting shellcode similar to the one used in existing detectors [9, 23, 38] can be enabled at will. Since the exact location of the shellcode in the input data is not known in advance, the emulator repeats the execution multiple times, starting from each and every position of the stream. In certain cases, however, the execution of some code paths can be skipped to optimize runtime performance [24].

The heuristics used in Gene are mostly based on memory accesses to certain locations in the address space of a vulnerable Windows process. To emulate correctly the execution of these accesses, the virtual memory of the emulator is initialized with an image of the complete address space of a typical Windows XP process taken from a real system. The image consists of 971 pages (4KB each), including the stack, heap, PEB/TIB, and loaded modules. All four heuristics use the same memory image and thus can be evaluated in parallel during execution.

Among other initializations before the beginning of a new execution [23], the segment register `FS` is set to the segment selector corresponding to the base address of the Thread Information Block, the stack pointer is set accordingly, while any changes to the original process image from the previous execution are reverted.

The runtime evaluation of the heuristics requires keeping some state about the occurrence of instructions with an operand that involved the `FS` register, as well as about read and write accesses to the memory locations specified in the heuristics. Regarding the SEH-based memory scanning heuristic (Sec. 3.2.1), although SEH chain validation is more complex compared to other instrumentation operations, it is triggered only if conditions S1 and S2 are true, which in practice happens very rarely.

When an `int 0x2e` instruction is executed, the `eax` register is checked for a value corresponding to one of the system calls that can be used for memory scanning, as described in Sec. 3.2.2. Although the actual functionality of the system call is not emulated, the proper return value is stored in the `eax` register depending on the validity of the supplied memory address. In case of an egg-hunt shellcode, this behavior allows the scanning loop to continue normally, resulting to several system call invocations.

5. EXPERIMENTAL EVALUATION

5.1 Detection Effectiveness

We began our evaluation with the shellcodes contained in the Metasploit Framework [2]. For Windows targets, Metasploit includes six basic payloads for downloading and executing a file, spawning a shell, adding a user account, and so on, as well as nine "stagers." In contrast to an egg-hunt shellcode, which searches for a second payload that has already been injected into the vulnerable process along with the egg-hunt shellcode, a stager establishes a channel between the attacking and the victim host for uploading other second-stage payloads. We generated plain (i.e., non-encrypted) instances of the above 15 shellcodes, as well as another 15 polymorphic instances of the same shellcodes using the ShikataGaNai encoder. As shown in Fig. 5, both Gene and the GetPC-based heuristic detected the polymorphic versions of the shellcodes. However, the original (plain) versions do not exhibit any self-decrypting behavior and are thus detected only by Gene. For both plain and polymorphic versions, Gene identified the shellcode using the PEB heuristic. The use of the PEB-based method for locating `kernel32.dll` is probably preferred in Metasploit due to its reliability.

We continued our evaluation with 22 samples downloaded from the shellcode repository of the Nepenthes Project [6]. Two of the samples had a broken decryptor and could not be executed properly. By manually unpacking the two payloads and scanning them with Gene, in both cases the shellcode was identified by the PEB heuristic. From the rest 20 shellcodes, 16 were identified by the PEB heuristic, and one, named "Saalfeld," by the SEH heuristic. The Saalfeld shellcode is of particular interest due to the use of a custom SEH handler although it is not an egg-hunt shellcode. The SEH handler is registered for safely searching the address space of the vulnerable process starting from address 0x77E00000, with the aim to reliably detect the base address of `kernel32.dll`. The SEH heuristic identifies the proper registration of a custom SEH handler, so the shellcode was successfully identified.

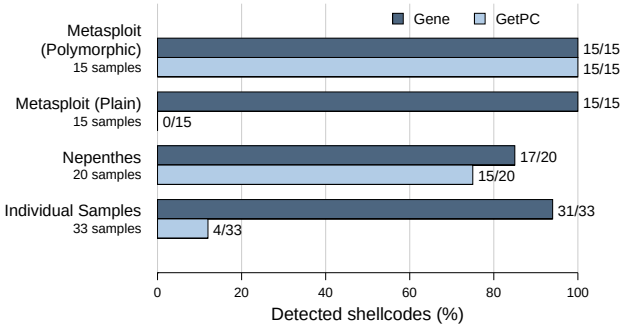


Figure 5: Number of shellcodes detected by Gene and the existing GetPC-based heuristic [9, 23, 38] for different shellcode sets. From a total of 83 different shellcode implementations, Gene detected 78 samples (94%), compared to 34 (41%) for the GetPC heuristic.

The remaining three shellcodes were missed due to the use of hard-coded addresses, e.g., the linear address of `kernel32.dll`, instead of reliable base address resolution. It would be trivial to implement another detection heuristic similar to the PEB heuristic based on commonly used hard-coded addresses in place of addressing based on the `FS` register to detect this kind of shellcode. However, these samples correspond to quite old attacks and this style naively implemented kind of shellcode is now encountered rarely. From the 20 shellcodes, 15 are self-decrypting and are thus detected by the GetPC-based heuristic.

Besides a few proof-of-concept implementations [5, 27] which are identified correctly by Gene, we were not able to find any other shellcode samples that locate `kernel32.dll` using backwards searching, probably due to the simplicity of the alternative PEB-based technique. In addition to the Saalfeld shellcode, the SEH heuristic detected a proof-of-concept SEH-based egg-hunt implementation [28], as well as the "omelet" shellcode [36], an egg-hunt variation that locates and recombines multiple smaller eggs into the whole original payload. The SEH heuristic was also effective in detecting polymorphic shellcode that uses SEH-based GetPC code [30], which is currently missed by existing payload execution systems. The SYSCALL heuristic was tested with three different egg-hunt shellcode implementations [27,28,37], which were identified correctly. In addition to these eight shellcode implementations, we gathered more Windows shellcode samples from public repositories [1, 3, 4], totaling 33 different samples. As shown in Fig. 5, the GetPC-based heuristic detected only four of the shellcodes that use simple XOR encryption, while Gene detected all but two of the samples, again due to the use of hard-coded addresses.

Finally, as an extra verification experiment, we tested Gene with a large dataset of real polymorphic attacks captured in production networks by Nemu [22]. Without using any self-decryption heuristic, this data set allows us to test the effectiveness of Gene in iden-

tifying the actual plain shellcode after the decryption process has completed. Gene analyzed more than 1.2 million attacks, which after the decryption process resulted to 98,602 unique payloads, and in all cases it identified the decrypted plain shellcode correctly. Not surprisingly, all shellcodes were identified by the PEB heuristic.

5.2 Heuristic Robustness

5.2.1 False Positives Evaluation

We tested the robustness of the heuristics against false positives using a large and diverse set of benign inputs. For our first experiment, we captured the internal and external traffic in two research and educational networks and kept the client-initiated stream of each TCP flow, since currently Gene detects only attacks against network services. Collectively, the data set consists of 15.5 million streams, totaling more than 48GB of data. Depending on its size, a stream can have from a few hundreds to many thousands of valid instruction sequences which are all analyzed independently by Gene. Thus, we consider as a false positive any benign input with at least one instruction sequence that matches one of the heuristics. When scanning the 15.5 million streams of this data set with Gene, none of the inputs matched any of the heuristics, resulting to zero false positives.

Seeking more evidence for the resilience of the heuristics against false positives, we continued the experiments with a much larger set of artificially generated benign data. The purpose of this experiment is to ensure that the random IA-32 machine code that is derived by interpreting arbitrary data as code does not match any of the heuristics. For this purpose, we used a script that continuously generates inputs of random binary and ASCII data that are subsequently scanned by Gene. The script generated 20 million 32KB-inputs of each type, totaling more than 1.3TB of data. The rationale behind using inputs consisting of random ASCII characters, in addition to random binary data, is to approximate the random code found in network streams that use text-based protocols. Similarly to the previous experiment, the false positive rate was again kept at zero.

5.2.2 Heuristic Analysis

We repeated the experiments of the previous section with the aim to explore in depth the behavior of the heuristics when operating on benign data. This time we measured the number of inputs with at least one instruction sequence that matched the first, the first two, or all three conditions of a heuristic.

Figure 6(a) shows the percentage of network streams that matched a given number of conditions. Out of 15.5 million inputs, only 82 (0.0005%) had an instruction sequence with a memory access to `FS:[0x30]` through the `FS` register—satisfying the first condition of the PEB heuristic. There were no streams that matched both the first and the second or all three conditions, which is a promising indication for the robustness of the PEB heuristic since all three conditions must be true for flagging an input as shellcode. The SYSCALL heuristic had a similar behavior, with just 51 of the inputs (0.0003%) exhibiting a single system call invocation, while there were no streams with two or more system calls.

A much larger number of streams matched the first condition of the BACKWD and SEH heuristics (8,620 and 41,063 streams, respectively). In both heuristics, the first condition includes a memory access to `FS:[0]`, which seems to appear more frequently in random code compared to accesses at `FS:[0x30]`. A possible explanation for this is that the effective address computation in the memory operand of some instruction can result to zero with a higher probability compared to other values. For example, when a `mov ebx,fs:[eax]` instruction is executed, it is more likely that `eax` will have been zeroed out, e.g., due to a previous two-byte long `xor eax,eax` instruction, instead of being set to 0x30. However, the percentage of inputs that matched both the first and the second condition is very low (0.0003% and 0.0004%, respectively), and no inputs matched all three conditions.

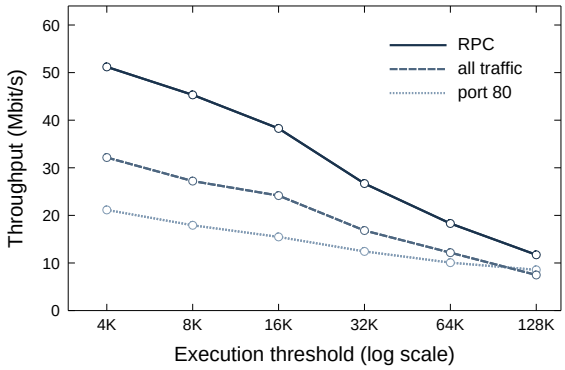


Figure 7: The raw processing throughput of Gene for different execution thresholds.

As shown in Fig. 6(b), the overall behavior when operating on random binary data is comparable to that for network streams, with no inputs fully matching any of the heuristics. However, for ASCII data (Fig. 6(c)), although the first condition in the PEB, BACKWD, and SEH heuristics matched in roughly 0.03% of the inputs, there were no inputs matching any of the subsequent conditions. The opcode for the `int` instruction falls outside the ASCII range, so no input matched not even the first condition of the SYSCALL heuristic. Overall, all heuristics seem to perform even better when operating on ASCII data.

5.3 Runtime Performance

We evaluated the processing throughput of Gene using the real network traffic traces presented in Sec. 5.2.1. Gene was running on a system with a Xeon 1.86GHz processor and 2GB of RAM. Figure 7 shows the raw processing throughput of Gene for different execution thresholds. The throughput is mainly affected by the number of CPU cycles spent on each input. As the execution threshold increases, the achieved throughput decreases because more emulated instructions are executed per stream. A threshold in the order of 8–16K instructions is sufficient for the detection of plain as well as the most advanced polymorphic shellcodes [24]. For port 80 traffic, the random code due to ASCII data tends to form long instruction sequences that result to degraded performance compared to binary data.

The overall runtime throughput is slightly lower compared to existing emulation-based detectors [23, 24] due to the overhead added by the virtual memory subsystem, as well as because Gene does not use the zero-delimited chunk optimization used in these systems [23]. Previous approaches skip the execution of zero-byte delimited regions smaller than 50 bytes, with the rationale that most memory corruption vulnerabilities cannot be exploited if the attack vector contains null bytes. However, the detection heuristics of Gene can identify shellcode in other attack vectors that may contain null bytes, such as document files. Furthermore, our approach can be applied in other domains [14, 15], for example for the detection of client-side attacks, in which the shellcode is usually encrypted at a higher level using some script language, and thus can be fully functional even if it contains null bytes.

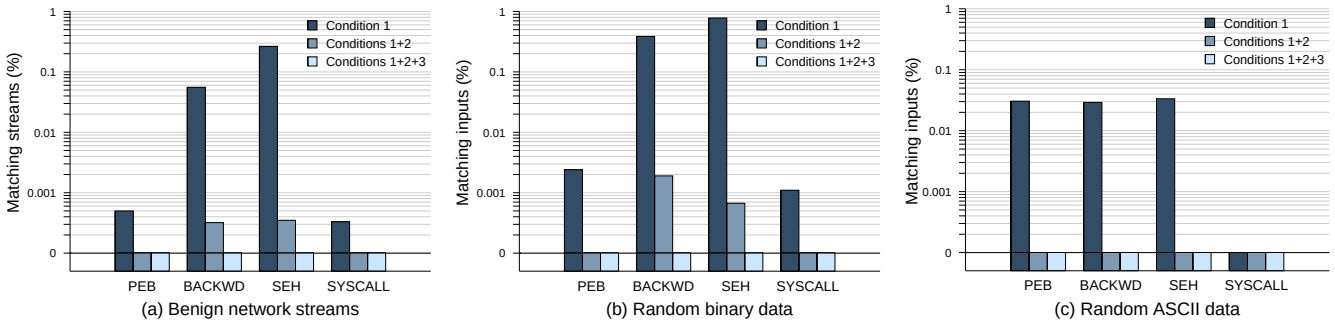


Figure 6: False positives evaluation with (a) 15.5 million real network streams (48GB total data size), (b) 20 million randomly generated binary inputs (650GB), and (c) 20 million randomly generated ASCII inputs (650GB). For all heuristics, none of the inputs matched all three conditions, resulting to zero false positives.

In practice, Gene can monitor high speed links when scanning for server-side attacks because client-initiated traffic (requests) is usually a fraction of the server-initiated traffic (responses). In our preliminary deployments in production networks, Gene can scan traffic of up to 100 Mbit/s without dropping packets. Furthermore, Gene currently scans the whole input blindly, without any knowledge about the actual network protocol used. Augmenting the inspection engine with protocol parsing would significantly improve the scanning throughput by inspecting each protocol field separately.

5.4 Real-world Deployment

We have deployed Gene in two University networks, where it has been operational since 25 November 2009. In these two deployments, Gene scans the traffic between the internal network and the Internet, as well as the traffic between selected internal subnets. As of 17 April 2010, Gene has detected 116,513 code injection attacks against internal and external hosts in these two networks. Although we cannot know how many of the attacks actually infected the targeted host, since many systems might had been previously patched, in all cases the attacker was able to connect and send the malicious input to the potentially vulnerable service. Almost one third of the attacks were launched from internal PCs, probably already infected by malware. About 86% of the attacks targeted port 445, while there were also attacks against ports 80, 135, 139, and 2967.

In both deployments, Gene uses the four new heuristics presented in this paper, as well as the GetPC heuristic used in existing polymorphic shellcode detectors, allowing us to compare the detection coverage of both approaches. The PEB heuristic matched in all of the attacks, supporting the fact that this is the most widely used technique for resolving `kernel32.dll`. However, the GetPC heuristic was triggered only by 85,144 attacks, i.e., 31,369 attacks (27%) did not use any form of self-decrypting shellcode. This means that the ability of Gene to detect plain shellcode increased the detection coverage for server-side code injection attacks by 37% compared to existing polymorphic shellcode detection approaches. By statically analyzing the identified machine code [22] we confirmed that in all cases it corresponds to actual shellcode, and so far we have not encountered any false positives.

6. DISCUSSION

The runtime heuristics presented in this paper allows Gene to detect a broad range of different shellcode classes. Of course, we cannot exclude the possibility that there are other kinds of Windows shellcode, or alternative techniques to those on which the heuristics are based, that may have missed our attention or have not been publicly released yet. Nevertheless, the architecture of Gene allows the parallel evaluation of multiple heuristics, and thus the detection engine can be easily extended with more heuristics for other shellcode types. For example, for our experimental evaluation, we have already implemented a fifth heuristic based on the widely used GetPC code technique used in existing polymorphic shellcode detectors [23, 24, 38]. In our future work, we plan to implement heuristics for the detection of the code required in a swarm attack [13], Linux-specific plain shellcode, Windows shellcode that uses hard-coded addresses, and so on.

A well known evasion technique against dynamic code analysis systems is the use of very long loops that force the detector to spend countless cycles until reaching the execution threshold, before any signs of malicious behavior are shown [32]. Gene uses infinite loop squashing [23] to reduce the number of inputs that reach the execution threshold. As stated in the literature [23, 24], the percentage of inputs with an instruction sequence that reaches the execution threshold ranges between 3–6%, which we also verified during the experiments of this paper. Since this is a small fraction of all inspected inputs, the endless loops in these sequences can potentially be analyzed further at a second stage using other techniques such as static analysis or symbolic execution [25].

Another inherent limitation of emulation-based shellcode detection is the lack of an accurate view of the system's state at the time the injected code would run on the victim system. This information includes the values of the CPU registers, as well as the complete address space of the particular exploited process [10, 23]. Although register values can sometimes be inferred [24], and Gene augments the emulator with the complete address space of a typical Windows process, which includes the most common system DLLs used by Windows shellcode, the shellcode may perform memory accesses to application-specific DLLs that are not known in advance, and thus cannot be followed by the emulator [16]. Fortunately, when protecting specific services, exact memory images of each service can be used in place of the generic process image. However, as already discussed, since the linear addresses of DLLs change quite often across different systems, and due to the increasing adoption of address space layout randomization and DLL rebasing, the use of absolute addressing results to less reliable shellcode. On the other hand, when the emulator runs within the context of a protected application, as for example in the browser-embedded detector proposed by Egele et al. [14], the emulator can have full access to the complete address space of the process.

Some of the operations matched by the heuristics, such as the registration of a custom exception handler, might also be found in legitimate executables. However, Gene is tailored for scanning inputs that otherwise should not contain executable IA-32 code. In case of file uploads, Gene can easily be extended to identify and extract executable files by looking for executables' headers in the inspected traffic, and then pass them on to a virus scanner.

7. RELATED WORK

Having realised the limitations of signature-based approaches in the face of polymorphic code and targeted attacks, several research efforts turned to static binary code analysis for identifying the presence of shellcode in network streams. One of the first such approaches by Toth and Kruegel uses code disassembly on network streams to identify the NOP-sled that sometimes precedes the shellcode [33]. Focusing on the shellcode itself, Anderson et al. [8] propose to scan each input for multiple occurrences of instruction sequences that end with an `int 0x80` instruction for the identification of Linux shellcode, with the rationale that the shellcode will have to execute several system calls. Other detection methods that use static code analysis aim to detect previously unknown polymorphic shellcode based on the identification of structural similarities among different worm instances [17], control and data flow analysis [12, 34, 35], or neural networks [20].

However, methods based on static analysis can be easily evaded by malicious code that uses obfuscation methods such as indirect jumps and self-modifications [23], which are widely used by current malware packers and polymorphic shellcode engines. In contrast, emulation-based detection can correctly handle even extensively obfuscated malicious code [23]. Polychronakis et al. propose the use of code emulation for the detection of self decrypting shellcode at the network level [23, 24]. The detection algorithm is based on the identification of the GetPC code and the self-references that take place during the execution of the shellcode. Zhang et al. propose to combine network-level emulation with static and data flow analysis for improving the runtime performance of the GetPC heuristic [38].

`Libemu` [9] is an open-source x86 emulation library tailored to shellcode analysis and detection. Shellcode execution is identified using the GetPC heuristic. `Libemu` can also emulate the execution of Windows API calls by creating a minimalistic process environment that allows the user to install custom hooks to API functions. Although the actual execution of API functions can be used as an indication for the execution of shellcode, these actions will be observed only after `kernel32.dll` has been resolved and the required API functions have been located through the EDT or IAT. Compared to the `kernel32.dll` resolution heuristics presented in Section 3.1, this technique would require the execution of a much larger number of instructions until the first API function is called, and also the emulation of the actual functionality of each API call thereafter. This means that the execution threshold of the detector should be set much higher, resulting to degraded runtime performance. For applications in which the emulator can spend more cycles on each input, both heuristics can coexist and operate in parallel, e.g., along with all other heuristics used in Gene, offering even better detection accuracy.

Besides the detection of code injection attacks against network services [22], emulation-based shellcode detection using the GetPC heuristic has been used for the detection of drive-by download attacks and malicious web sites. Egele et al. [14] propose a technique that uses a browser-embedded CPU emulator to identify javascript string buffers that contain shellcode. Wepawet [15] is a service for web-based malware detection that scans and identifies malicious web pages based on various indications, including the presence of shellcode. The CPU emulator in both projects is based on `libemu`.

Shellcode analysis systems help analysts study and understand the structure and functionality of a shellcode sample. Ma et al. [18] used code emulation to extract the actual runtime instruction sequence of shellcode samples captured in the wild. Spector [11] uses symbolic execution to extract the sequence of library calls made by the shellcode, along with their arguments, and at the end of the execution generates a low-level execution trace. Yataglass [25] improves the analysis capabilities of Spector by handling shellcode that uses memory-scanning attacks.

8. CONCLUSION

The increasing professionalism of cyber criminals and the vast number of malware variants and malicious websites make the need for effective code injection attack detection a critical challenge. To this end, shellcode detection using payload execution offers important advantages, including generic detection without exploit or vulnerability-specific signatures, practically zero false positives, while it is effective against targeted attacks.

In this paper we present a comprehensive shellcode detection method based on code emulation. Our approach expands the range of malicious code types that can be detected by enabling the parallel evaluation of multiple runtime heuristics that match inherent low-level operations during the execution of different shellcode types. The runtime heuristics presented in this work enable the effective detection of plain and metamorphic shellcode, both of which are not identified by existing shellcode detectors. This is achieved regardless of the use of self-modifying code or dynamic code generation, on which existing emulation-based polymorphic shellcode detectors are exclusively based.

Our experimental evaluation shows that the proposed approach can effectively detect a broad range of diverse shellcode types and implementations, increasing significantly the detection coverage compared to existing emulation-based detectors, while extensive testing with a large set of benign data did not produce any false positives. Gene, our prototype implementation of the proposed technique for the detection of server-side code injection attacks detected 116,513 attacks against production systems in a period of almost five months without false positives.

Although Gene currently operates at the network level, the proposed detection heuristics can be readily implemented in emulation-based systems in other domains, including host-level or application-specific detectors. As part of our future work, we plan to implement more heuristics to cover the detection of less widely used shellcode types, such as shellcode that uses hard-coded addresses, and explore the design of a description language that would expedite the development of new heuristics.

Acknowledgments

We would like to thank Periklis Akritidis and Angelos Keromytis for their valuable feedback on earlier versions of this paper. This work was supported in part by the Marie Curie FP7-PEOPLE-2009-IOF project MALCODE and the FP7 project SysSec, funded by the European Commission under Grant Agreements No. 254116 and No. 257007, and by the project i-Code, funded by the Prevention, Preparedness and Consequence Management of Terrorism and other Security-related Risks Programme of the European Commission—Directorate-General for Home Affairs under Grant Agreement No. JLS/2009/CIPS/AG/C2-050. This publication reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained herein. Michalis Polychronakis is also with FORTH-ICS. Evangelos Markatos is also with the University of Crete.

9. REFERENCES

[1] Goodfellas security research team. http://goodfellas.shellcode.com.ar/.

[2] The metasploit project. http://www.metasploit.com/.

[3] milw0rm. http://milw0rm.com/shellcode/win32/.

[4] Packet storm. http://www.packetstormsecurity.org/.

[5] Win32 assembly components, Dec. 2002. http://lsd-pl.net.

[6] Common shellcode naming initiative, 2009. http://nepenthes.carnivore.it/csni.

[7] Retrieving kernel32's base address, June 2009. http://www.harmonysecurity.com/blog/2009/06/retrieving-kernel32s-base-address.html.

[8] S. Andersson, A. Clark, and G. Mohay. Network-based buffer overflow detection by exploit code analysis. In *Proceedings of the Asia Pacific Information Technology Security Conference (AusCERT)*, 2004.

[9] P. Baecher and M. Koetter. libemu, 2009. http://libemu.carnivore.it/.

[10] P. Bania. Evading network-level emulation, 2009. http://piotrbania.com/all/articles/pbania-evading-nemu2009.pdf.

[11] K. Borders, A. Prakash, and M. Zielinski. Spector: Automatically analyzing shell code. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2007.

[12] R. Chinchani and E. V. D. Berg. A fast static analysis approach to detect exploit code inside network flows. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2005.

[13] S. P. Chung and A. K. Mok. Swarm attacks against network-level emulation/analysis. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2008.

[14] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of the 6th international conference on Detection of Intrusions and Malware, & Vulnerability Assessment (DIMVA)*, 2009.

[15] S. Ford, M. Cova, C. Kruegel, and G. Vigna. Wepawet, 2009. http://wepawet.cs.ucsb.edu/.

[16] I)ruid. Context-keyed payload encoding. *Uninformed*, 9, Oct. 2007.

[17] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2005.

[18] J. Ma, J. Dunagan, H. J. Wang, S. Savage, and G. M. Voelker. Finding diversity in remote code injection exploits. In *Proceedings of the 6th Internet Measurement Conference (IMC)*, 2006.

[19] J. Mason, S. Small, F. Monrose, and G. MacManus. English shellcode. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS)*, 2009.

[20] U. Payer, P. Teufl, and M. Lamberger. Hybrid engine for polymorphic shellcode detection. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 19–31, July 2005.

[21] M. Pietrek. A crash course on the depths of Win32™structured exception handling, 1997. http://www.microsoft.com/msj/0197/exception/exception.aspx.

[22] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. An empirical study of real-world polymorphic code injection attacks. In *Proceedings of the 2nd USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET)*, April 2009.

[23] M. Polychronakis, E. P. Markatos, and K. G. Anagnostakis. Network-level polymorphic shellcode detection using emulation. In *Proceedings of the Third Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, July 2006.

[24] M. Polychronakis, E. P. Markatos, and K. G. Anagnostakis. Emulation-based detection of non-self-contained polymorphic shellcode. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2007.

[25] M. Shimamura and K. Kono. Yataglass: Network-level code emulation for analyzing memory-scanning attacks. In *Proceedings of the 6th international conference on Detection of Intrusions and Malware, & Vulnerability Assessment (DIMVA)*, 2009.

[26] sk. History and advances in windows shellcode. *Phrack*, 11(62), July 2004.

[27] Skape. Understanding windows shellcode, 2003. http://www.hick.org/code/skape/papers/win32-shellcode.pdf.

[28] Skape. Safely searching process virtual address space, 2004. http://www.hick.org/code/skape/papers/egghunt-shellcode.pdf.

[29] SkyLined. Finding the base address of kernel32 in Windows 7. http://skypher.com/index.php/2009/07/22/shellcode-finding-kernel32-in-windows-7/.

[30] SkyLined. SEH GetPC (XP SP3), July 2009. http://skypher.com/wiki/index.php/Hacking/Shellcode/Alphanumeric/ALPHA3/x86/ASCII/Mixedcase/SEH_GetPC_(XP_sp3).

[31] Y. Song, M. E. Locasto, A. Stavrou, A. D. Keromytis, and S. J. Stolfo. On the infeasibility of modeling polymorphic shellcode. In *Proceedings of the 14th ACM conference on Computer and communications security (CCS)*, 2007.

[32] P. Ször. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, February 2005.

[33] T. Toth and C. Kruegel. Accurate buffer overflow detection via abstract payload execution. In *Proceedings of the 5th Symposium on Recent Advances in Intrusion Detection (RAID)*, Oct. 2002.

[34] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Still: Exploit code detection via static taint and initialization analyses. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2008.

[35] X. Wang, C.-C. Pan, P. Liu, and S. Zhu. Sigfree: A signature-free buffer overflow attack blocker. In *Proceedings of the USENIX Security Symposium*, Aug. 2006.

[36] B.-J. Wever. SEH Omelet Shellcode, 2009. http://code.google.com/p/w32-seh-omelet-shellcode/.

[37] G. Wicherski. Win32 egg search shellcode, 33 bytes. http://blog.oxff.net/2009/02/win32-egg-search-shellcode-33-bytes.html.

[38] Q. Zhang, D. S. Reeves, P. Ning, and S. P. Lyer. Analyzing network traffic to detect self-decrypting exploit code. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2007.

Cross-Layer Comprehensive Intrusion Harm Analysis for Production Workload Server Systems

Shengzhi Zhang
Department of Computer Science and Engineering, Pennsylvania State University, University Park, USA
suz116@psu.edu

Xiaoqi Jia
State Key Laboratory of Information Security, Institute of Software, Chinese Academy of Sciences, China
xjia@is.iscas.ac.cn

Peng Liu
College of Information Sciences and Technology Pennsylvania State University, University Park, USA
pliu@ist.psu.edu

Jiwu Jing
State Key Laboratory of Information Security, Graduate University of Chinese Academy of Sciences, China
jing@is.ac.cn

ABSTRACT

Analyzing the (harm of) intrusion to enterprise servers is an onerous and error-prone work. Though dynamic taint tracking enables *automatic* fine-grained intrusion harm analysis for enterprise servers, the significant runtime overhead introduced is generally intolerable in the production workload environment. Thus, we propose PEDA (Production Environment Damage Analysis) system, which decouples the onerous analysis work from the online execution of the production servers. Once compromised, the "has-been-infected" execution is analyzed during high fidelity replay on a separate instrumentation platform. The replay is implemented based on the *heterogeneous* virtual machine migration. The servers' online execution runs atop fast hardware-assisted virtual machines (such as Xen for near native speed), while the infected execution is replayed atop binary instrumentation virtual machines (such as Qemu for the implementation of taint analysis). From identified intrusion symptoms, PEDA is capable of locating the fine-grained taint seed by integrating the backward system call dependency tracking and one-step-forward taint information flow auditing. Started with the fine-grained taint seed, PEDA applies dynamic taint analysis during the replayed execution. Evaluation demonstrates the efficiency of PEDA system with runtime overhead as low as 5%. The real-life intrusion studies successfully show the comprehensiveness and the precision of PEDA's intrusion harm analysis.

Categories and Subject Descriptors

D.4.6 [**Operating System**]: Security and Protection—*Information flow controls*

General Terms

Security

Keywords

Cross-layer intrusion harm analysis, heterogeneous virtual machine migration, forward and backward tracking

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

1. INTRODUCTION

Upon system being compromised, a dilemma faced by enterprise security technicians is whether to aggressively continue the service for business continuity or to conservatively shut down the server for loss constrains. It can be even more complicated as whether to resume the service from a clean checkpoint regardless of accumulated system/services state or to pause the execution for a comprehensive clean-up. In this scenario, the right decision no doubt relies on a comprehensive intrusion harm analysis for the server systems, e.g., locating the intrusion "breakin" and identifying the intrusion "footprint" (infection and cascading effects caused by infection propagation). However, this basic yet essential task continues to bother the security technicians for years as an onerous and error-prone work. Hence, researchers' attention is caught by how to do automatic fine-grained intrusion harm analysis for production workload servers with concerns of precision (without losing fidelity) and performance (without slowing them down greatly). The recently proposed dynamic taint analysis can be applied to the servers' online execution to ensure the fidelity of intrusion analysis, while it intuitively causes significant runtime overhead (about 10-40X [21], [17], [15], [14] and [9]). Obviously, running the online server in that manner is not practical because business-critical production workload servers can't tolerate such overhead.

How to solve this problem in a practical way without losing fidelity depends on whether the following assumption is assumed true or not. The "taint seed" assumption: the taint seed is precisely located before the infection diagnosis

task starts. This assumption is indeed true in some particular cases. For example, when a remote exploit matches a newly generated signature after the intrusion event, the system would know which packet (seen during the past intrusion event) should be used as the taint seed. Under this assumption, solution to the above problem could be inspired from the fidelity-preserving whole machine replay idea proposed in [8] and [20]. However, the original replay technique cannot be directly applied because the replay required by fine-grained infection diagnosis can no longer be performed on the same online processing computer architecture "interface". Rather, the replay now needs to be performed on a binary instrumentation platform (such as QEMU). To tackle this challenge, we developed a heterogeneous VM (virtual machine) migration technique, which is quite different from the migration in Aftersight [6] as discussed in Section 7 (Note that PEDA and Aftersight are two independent works.).

But can we now really claim "mission accomplished"? Our answer is "perhaps not", which is based on the observation that the "taint seed" assumption in many other cases is not practical. Typically, the intrusion detection may often happen after attack escalation so that the intrusion symptoms reported by IDS (intrusion detection system) are not necessarily the "taint seed". For instance, IDS may report malicious system binary modification through integrity check, but the detected binary modification is obviously not the intrusion root, in other words, the "taint seed". To be able to solve the above problem without relying on that not-very-practical "taint seed" assumption, the following challenge must be tackled. The "seed-unknown" challenge: when only some (indirect) symptoms of the intrusion could be identified, how to do comprehensive fine-grained intrusion analysis? Existing backward system call dependency analysis Backtracking [11] can indeed identify the system-object-level intrusion root (typically a process) from the detected intrusion symptoms. However, directly treating the system-object-level intrusion root as taint seed for dynamic taint analysis will introduce much false positive, due to tainting the whole process address space and all its following operations.

In this work, besides solving the problem under the "taint seed" assumption through a heterogeneous VM migration technique, we take a novel approach to tackle the "seed-unknown" challenge. This approach integrates both the backward system call dependency analysis and the forward suspicious data flow analysis. Tracing the system call dependency graph backward can help us quickly identify the system-object-level intrusion root. Thereafter, we trace the identified system-object-level intrusion root to locate the one-step-down buffers that it uses to propagate the intrusion harm. In this way, the "seed-unknown" challenge can be addressed by treating those buffers as taint seed. Alternatively, we can also identify the malicious intrusion packets by tracing the processing flow of each suspicious network packet received by the system-object-level intrusion root. Then, the "seed-unknown" challenge can be addressed by treating the memory cells or disk sectors containing the malicious intrusion packets as taint seed. We have integrated these novel approaches into our heterogeneous VM migration solution. As a result, we get a rather complete and practical solution to do post-mortem fine-grained intrusion analysis for production workload servers under the "seed-unknown" assumption.

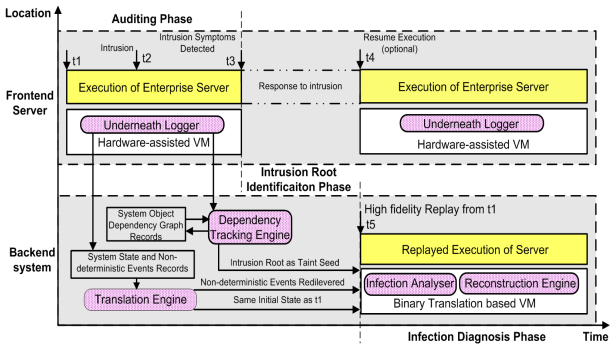


Figure 1: PEDA Architecture

The rest of this paper is organized as follows. Our PEDA approach is described in Section 2. The design of PEDA system with details of each functionality is presented in Section 3. In Section 4, we focus on the implementation issues of PEDA system. In Section 5, we evaluate PEDA in terms of efficiency, precision and comprehensiveness. Limitations and Related works are discussed in Section 6 and Section 7 respectively. Finally, we conclude the paper in Section 8.

2. PEDA APPROACH

The goal of PEDA system is to enable fine-grained intrusion analysis for enterprise-level production workload servers with precision and comprehensiveness. Hence, PEDA decomposes the intrusion analysis work into three phases: auditing phase, intrusion root identification phase and infection diagnosis phase. Figure 1 shows PEDA system architecture with five functional components: underneath logger, dependency tracking engine, translation engine, infection analyzer and reconstruction engine. Below, we briefly describe each phase with several components working in tandem to fulfil our desired functionality.

2.1 Auditing phase

During routine execution of enterprise servers, the underneath logger will periodically take a checkpoint of the whole server system, including disk, raw memory, CPU registers, RTC (real time clock), I/O devices, DMA, timers and etc. The whole checkpoint serves as the starting point for replay. Moreover, non-deterministic events, e.g. external inputs into the server such as network packets, keyboard inputs, timer interrupt and etc., between contiguous checkpoints are also recorded for redelivery during replay. The deterministic execution of the server systems (from the same initial state) and the non-deterministic events redelivery can ensure the high "fidelity" of the replay. This replay in turn helps PEDA to reveal "what had happened" since intrusion occurred. In order to tackle the "seed-unknown" challenge, during the routine execution of the server system, PEDA needs to record all the system operations that can cause potential dependency between system objects. Then, the dependency tracking engine operates on those system operations to dynamically generate system-object (file or process) dependency graph. Once intrusion symptoms detected, the graph is ready to be used to quickly identify the fine-grained intrusion root for dynamic taint analysis.

2.2 Intrusion root identification phase

This phase only works under the "seed-unknown" assumption. Whenever some intrusion symptoms, e.g., system binaries are suspiciously modified, are captured, the dependency tracking engine will start to trace these captured symptoms backward throughout the already-produced system-object dependency graph. This backward tracking can help us swiftly identify the system-object-level intrusion root, typically the network-oriented process. In order to identify the fine-grained taint seed for infection analyzer, the dependency tracking engine performs one-step-forward auditing to locate the buffers (used by the system-object-level intrusion root) containing the taint propagation data. Simultaneously, the translation engine translates the logging information recorded by the much faster hardware-assisted VM into the form that the analyzing binary translation based VM can "understand". Thus, the later-on replay can be done on a heterogeneous VM. All the work of dependency tracking engine and translation engine is done on the backend system, without incurring additional runtime overhead to the online servers.

2.3 Infection diagnosis phase

When the translation engine finishes the system states and non-deterministic events translation, the "has-been-infected" server's execution is ready to be replayed on the binary instrumentation platform with high fidelity. Started with the fine-grained taint seed either known directly or identified by dependency tracking engine, the infection analyzer performs the fine-grained instruction flow taint analysis. Both the data taint flow and the control taint flow are applied to prevent some intended attackers crafting code that can evade the data flow auditing. Generally, the fine-grained taint analysis can only generate instruction flow dependency, which contains valuable binary information but lacks operating system semantics. Therefore, reconstruction engine is also developed to bridge this kind of "semantic gap" [5] by dynamically mapping each instruction flow with system objects. Through the coordination of the infection analyzer and reconstruction engine, we can provide the cross-layer infection diagnosis results both at the system object layer (full-of-semantics) and the instruction layer (comprehensive).

3. DESIGN OF PEDA SYSTEM

In this section, we focus on the design of PEDA system by describing the details of analysis decoupling, heterogeneous VM migration, fine-grained intrusion root identification and cross layer infection diagnosis.

3.1 Analysis Decoupling

PEDA takes the idea of analyzing the intrusion during high fidelity replay instead of during the first run for the following reasons. First, replay-based intrusion analysis can take over the workload of fine-grained taint analysis from the routine execution of production workload servers. In this way, the performance of the server systems can be ensured, without incurring the 10-40x overhead introduced by taint analysis. Second, replay-based intrusion analysis can take advantage of the "already-happened" knowledge to reduce the assessment workload. For instance, intrusion root can be identified by logging system calls, generating dependency graph, and integrating IDS detected intrusion symptoms. During the first run, however, the auxiliary information is not available yet. Last, by decoupling the intrusion analysis off the main server, PEDA offers the flexibility for enterprise security technicians to either aggressively restart (or continue running) the service for business continuity (based on swift system-object-level intrusion propagation assessment) or conservatively shut down the server for loss constrains (waiting for comprehensive intrusion analysis). The decoupling is implemented by recording the whole system state and non-deterministic events during the routine execution of the server.

3.1.1 Checkpointing

PEDA periodically takes a snapshot of the whole server system to ensure that the replay shares the same initial state as the first run. Different checkpoints enable the system execution to be replayed at different time of the first run. This is feasible and realistic to analyze a specific event, such as intrusion, on a long-running server system, because there is no need to replay the system execution from the very beginning of the first run. The checkpoint contains all the hardware states, such as CPU registers, raw memory, disk, I/O device, timers, DMA and etc. A naive way to take a consistent checkpoint of the whole system is to pause the server system, take a snapshot, and then resume the execution. However, for production workload servers with large disk and raw memory, this "stopping-the-world" checkpointing will cause intolerable service downtime to the servers.

On the other hand, we observed that during 2:00 am to 5:00 am, the amount of service requests are much more degraded than that during daytime for Amazon-style servers. Hence, PEDA is designed to take checkpoint infrequently (e.g., once per day during the service degradation period). To further take advantage of the servers' working style, PEDA is designed to trade service response time off for service live time. In particular, PEDA initializes a pre-checkpointing phase, during which the disk and raw memory states are recorded with the server system "on the fly". Thereafter, PEDA pauses the server system and establishes a stop-and-copy phase. During this phase, PEDA records all the other device states and the changes to disk as well as memory since the start of pre-checkpointing phase. Finally, PEDA commits the end of checkpointing and resumes the execution of the server. By means of the pre-checkpointing phase, the heavy workload of storage checkpointing is taken over from the stop-and-copy phase, thus greatly reducing the service downtime.

3.1.2 Non-deterministic events logging

For the production workload servers, the non-deterministic events are mainly the service-requesting network packets, the administrator's management keyboard inputs, and the I/O devices' interrupts. Meanwhile, the keyboard inputs happen quite infrequently for such kind of servers. Thus, it will only introduce little runtime overhead to the servers to log them directly using emulated keyboard of virtual machine. However, this is not the case for network packets. Typically, the production workload server deals with thousands of or even more service-requesting packets everyday. Hence, it will introduce intolerable overhead to log these packets by emulated NIC (network interface card) of virtual machine, because the NIC needs to perform an additional data transfer per packet.

PEDA successfully solves this problem by leveraging a router to split the incoming packets and to forward them to

	qemu-dm	xen hvm
Devices	NIC, vga, keyboard, mouse, dma, timer, and etc.	CPU registers, memory, ioapic, apic, pic, pit, and pmtimer.

Figure 2: Xen Devices Emulation

both the target server and the backend server. The backend server will log the contents of all these packets. Simultaneously, the emulated NIC on target server will only record the header identification information of each packet. During the intrusion root identification phase, the translation engine will associate each logged packet with its identification information. All the I/O devices' interrupts to CPU are logged through the device emulation code of virtual machine. In order to exactly redeliver the interrupt during replay, we record the timing semantics at which the interrupt occurs. For instance, we log the time at which the keyboard input arrives, and the instruction at which the corresponding interrupt is delivered to CPU. The time is logged by the unit of CPU clicks, while the instruction is logged using the program counter and the number of branches executed [4] by means of one provided hardware performance counter.

3.2 Heterogeneous VM Migration

Typically, the instruction flow taint analysis needs to be implemented in the binary translation based VM. Thus, a direct way to do decoupled analysis is to let the online server run on top of such kind of VM during routine execution, and to migrate the recorded VM image onto such kind of VM with analyzing module when intrusion analysis is needed. However, the problem with this approach is the intolerable runtime overhead (3X-4X compared with native execution [3]) introduced to the production workload server by VM binary translation during routine execution. To minimize the runtime overhead, PEDA runs the server with underneath logger atop hardware-assisted near native-speed VM during routine execution. When intrusion analysis is needed, PEDA replays the recorded VM image atop binary-translation based VM to capture the infection propagation. In particular, PEDA implements heterogeneous VM migration functionality to help the latter (binary-translation based VM) "understand" the system states and events recorded by underneath logger in the former (hardware-assisted VM).

Generally, different virtual machines use different device emulation techniques, so "translating" the device state of one VM to that of the other VM is not an easy task. One way to evade such kind of device state translation is to only consider the output data flow from each device to CPU. Aftersight [6] applies this approach by directly recording the data flows from the devices to CPU and redelivering them to CPU during replay. However, performance is trade off for the "bypass" of the device emulation incompatibility. Logging raw data out of the device introduces much more runtime overhead than recording the external inputs to the device, especially for the large amount of data read from disk (As also noted in Aftersight [6], to replay a disk read operation, their method must record the actual data being read from the emulated disk.).

With much concern of production workload server's performance, PEDA takes the approach of directly recording the external inputs to devices, and leverages a translation engine to eliminate the device emulation incompatibility. PEDA simplifies the the whole system state translation/migration work by choosing Xen as hardware-assisted VM and Qemu as binary translation based VM. The reason is that Xen-HVM relies a lot on Qemu "device manager" (qemu-dm) daemon running backend in Domain 0 to provide device I/O emulation. Figure 2 shows various Xen devices emulated by qemu-dm. There exist several devices such as CPU registers, apic and etc., as shown in Figure 2, which are emulated by Xen-HVM itself. However, they share similar entries with those emulated by Qemu, and our implementation shows that the scaling will not consume much time for an experienced programmer. Although translation engine can eliminate the device emulation incompatibility, it cannot handle hardware diversities without adding new "translation rules", such as rtl8139 NIC and e1000 NIC.

3.3 Addressing the Seed-Unknown Challenge

Identifying the intrusion root is a critical step of intrusion analysis, because it determines where to patch the vulnerabilities and what to be audited during analysis. We cannot simply rely on IDS (Intrusion Detection System) to inform us the intrusion root, because intrusion symptoms notified by IDS often lag behind the actual intrusion breakin. PEDA identifies the system-object-level (processes or files) intrusion root in a similar way as Backtracking [11]. However, the system-object-level intrusion root cannot be provided to infection analyzer as taint seed, because dynamic taint tracking requires the taint seed at the granularity of memory cell or disk segment. For instance, if taking a process as taint seed, then all the operations and the whole address space of this process should be tainted since it was compromised. This will generally result in taint explosion [16] throughout the server system with high false positive, thus hurting not only the efficiency but also the precision (correctness) of our infection diagnosis. Hence, PEDA implements one-step-forward auditing to "dip" further down to the memory buffer and identifies the fine-grained intrusion root there. As shown in Figure 1, PEDA relies on the dependency tracking engine to do both the system-object-level dependency graph generation during the auditing phase and fine-grained taint seed identification during the intrusion root identification phase.

3.3.1 Dependency Graph Generation

We specify system object dependency as a source object, a destination object, and a specific time. For instance, if one process reads a file, then the file is the source object; the process is the destination object; while the time is defined as when the process issues the *read* system call. PEDA records the process id issuing the system call, parameters of that system call, and the system call issuing time or sequence. Then they are associates with source object, destination object and time respectively. Since system objects are generally processes and files, we define two categories of system-object dependency: process/process dependency and process/file dependency.

Process/Process Dependency Whenever one process affects the operation of the other process, we should mark these two processes as dependency relationship. The system call issuer should be marked as source object and the process identity specified in the system call parameters should be marked as destination object. The time t should be set as

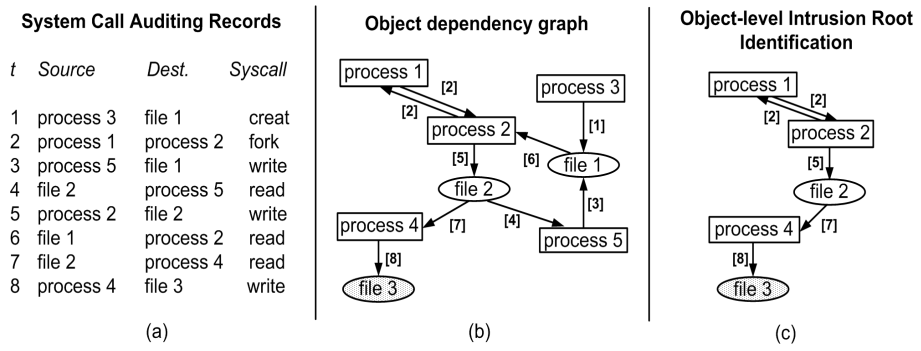


Figure 3: Dependency Tracking Engine Functionality

the system call issuing time or sequence. Note the creation of child process through issuing *clone* system call. We will apply bi-directional dependency between parent and child process during the period of address space sharing, because the parent process and the child process will share the parent's address space until the child process issues *execv* system call.

Process/File Dependency One process may create, open, read, write, delete files, or change the owner, property of the files, and etc. The dependency between process and file should be established once any of these operations takes effect. The determination of source object and destination object depends on the actual data flow as shown in Figure 3(b). Generally, the time t should be set as the system call issuing time or sequence. However, one process can map one file to its address space by issuing *mmap* system call, and then read/write the file directly by using store/load instructions, which can no longer be captured by system call level auditing. In this scenario, we will maintain the dependency from the time when the *mmap* is issued to the time when the corresponding *munmap* is issued.

Taking the system call auditing records as input, dependency tracking engine can generate system object dependency graph dynamically during auditing phase. Figure 3(b) shows a segment of the dependency graph based on the system call records in Figure 3(a). Each node denotes a system object either as source or destination. The $\xrightarrow{[t]}$ denotes the dependency relationship between the source object and the destination object at time t. Since the graph may grow quite large and produce false positive results on taint propagation, PEDA performs graph pruning to reduce the storage size and false dependencies. For instance, we do not consider situations like independent process termination, irrelevant signals, or accessing dummy objects like *stdin/stdout* or */dev/null*.

3.3.2 Coarse-grained and Fine-grained Intrusion Root Identification

When any intrusion symptom is detected by IDS, e.g., some maliciously modified system binaries through integrity check, the dependency tracking engine switches to intrusion root identification mode. It starts tracing the system-object-level dependency graph backward from the detected intrusion symptoms. The system-object-level intrusion root identification is already studied by *Backtracking* [11]. Here, we adopt a similar approach. Figure 3(c) shows the system-object-level intrusion root identification results based on dependency graph segment in Figure 3(b). We locate the detected infected objects from the dependency graph, trace the dependency chain back with timestamps, and eliminate uninfected objects from the graph. For production workload servers with constrains of physical access, the intrusion breakin should mainly occur at the network-service-oriented applications. Therefore, we trace back the intrusion propagation flow, locate the very beginning network-oriented process, and identify it as the system-object-level intrusion root. In addition, system security technicians can also specify a set of vulnerable services and ports to further refine this procedure. PEDA also records the intrusion propagation timestamp when the intrusion root object performs the first operation that eventually propagates to the detected intrusion symptoms.

In order to bridge the gap between the system-object-level intrusion root and the instruction flow taint tracking, we develop a straightforward but effective method, one-step-forward auditing, to locate the fine-grained intrusion root for infection analyzer. Once we have identified the system-object-level intrusion root (generally network-oriented process), we examine all the system calls issued by it. By analyzing the parameters of these system calls, we extract the ones propagating the intrusion to the detected intrusion symptoms. In this way, we can locate the buffers that actively expand the infection, i.e., from the intrusion root object to the one-step-down objects in the intrusion propagation graph. Taking these buffers as fine-grained taint seed will provide us the precise and comprehensive infection diagnosis.

There are several drawbacks for the one-step-forward-auditing approach to identify the fine-grained intrusion root. First, it cannot provide any information regarding how the intrusion root object is compromised, so there is no way for the infection analyzer to trace the intrusion breakin from the very beginning. Second, it relies on an implicit assumption that the backward tracking can extract at least all the one-step-away infected objects from intrusion root in the dependency graph. This can be generally true, but some intended attackers aware of backward tracking could craft intrusion programs to evade such kind of auditing.

Therefore, we provide an alternative way to identify the fine-grained intrusion root. We take advantage of the general belief that the intrusion breakin should start from the network packets, and try to associate some packets with the infection propagation. If any packet processing information flow finally "contributes" to the infection propagation, we

provide the receiving buffer or storing disk sectors of this packet to the infection analyzer as taint seed. However, tracking the everyday thousands of packets to production workload server is generally infeasible.

Fortunately, we have already identified the system-object-level intrusion root with intrusion propagation timestamp, so we have a rough idea of where (the intrusion root object) and when (before the timestamp) the intrusion breaks into the victim system. Thus, we only need to track those network packets sent to this intrusion root object (generally an application) before the intrusion propagation timestamp. Furthermore, it is feasible for system security technicians to refer to the server firewall's "whitelist" to filter the packets from trustable remote identities. Finally, we can start multiple packet-auditing instances simultaneously on separate analysing instrumentation platforms to further increase the efficiency.

After a much smaller set of suspicious packets is identified, we use the following technique to determine the actual intrusion packets. The dependency tracking engine can leverage the translation engine to replay the server system execution. During the high fidelity replay, it audits the processing information flow of those packets by applying instruction flow taint tracking since they entered the receiving buffers. If any packet is manipulated by the intrusion root object and hits the system-object-level intrusion propagation chains, then the corresponding packet should be considered as the intrusion packet. Note that several packets instead of one may contribute to one single intrusion. Generally, when one intrusion packet has been identified, we can rely on the identification information contained in the header of that packet to swiftly locate other intrusion packets if any. Once all the intrusion packets are identified, we specify the buffers or the disk sectors storing those packets as fine-grained intrusion root, i.e., taint seed.

3.4 Cross Layer Infection Diagnosis

The infection analyzer is implemented in the binary translation based, whole system emulator Qemu [3]. With an abstract view of the hardware such as CPU registers, memory, and I/O devices, it enables down-to-byte comprehensive infection diagnosis by auditing the emulated hardware. However, the "semantic gap" [5] really exists because it lacks the meaningful information of operating system semantics. Thus, we also develop reconstruction engine to present the infection diagnosis results both at the system object layer and at the instruction layer.

Our infection analyzer works similarly to several existing systems using dynamic taint analysis [21], [13] and [7]: auditing each instruction executed. Essentially, we capture the data flow dependences between instructions and certain control flow dependences such as *switch* and *if-else* statements. We implement the reconstruction engine totally at the VMM (virtual machine monitor) level without any interference to the guest OS. Our approach is also different from static analysis of raw memory and *sysmap* reconstruction ([10] and [18]). Our reconstruction engine extracts the system object semantics directly from CPU registers and dynamically maps the kernel address space with kernel data structures. Though this renders the reconstruction work a little more difficult, we can ensure the correctness of replay and the security of the PEDA components.

4. IMPLEMENTATION ISSUES OF PEDA SYSTEM

We implement PEDA prototype on Qemu-0.9.0 and Xen-3.3.0 to demonstrate its capability of comprehensive intrusion analysis for production workload servers. On both of them, we run the same image file with Linux kernel version 2.6.20 as Guest OS. The goal and design of PEDA system pose various challenges, hence we discuss the implementation issues of PEDA system in the rest of this section.

4.1 Checkpointing and Non-deterministic Event Logging

On Xen Domain 0 management console, we implement a new command *xm checkpoint*. Once issued, *xm checkpoint* is passed to *Xend* via *XML RPC*. *Xend* responses to this checkpoint request by initializing a pre-checkpointing phase, during which *Xend* coordinates with Xen hypervisor to start recording raw memory and virtual disk contents. Moreover, the hypervisor makes a shadow copy of all the following "writes" to memory and disk. Then, *Xend* pauses the system execution and establishes a stop-and-copy phase. During the stop-and-copy phase, the hypervisor records CPU registers, interrupt controllers and etc. Simultaneously, *Xend* commits all the shadow copy of "writes" to the memory and virtual disk recorded during previous phase, and calls qemu-dm to log other devices states such as NIC, VGA, keyboard, DMA and etc. Thereafter, *Xend* calls qemu-dm to start auditing the following keyboard inputs, network header identification information, and their arrival time at the unit of CPU clicks. The router is also notified to start directing the following network packets to both the server and the backend system separately. The backend system is modified to be able to receive and record these redirected packets. In addition, the hypervisor will activate the system events auditing functionality to record all the following system calls of the server system execution for dynamic dependency tracking. The whole system states from checkpoint together with keyboard inputs, packet identity, timing and system call records during the system execution are transferred to backend system through Gigabit Ethernet.

4.2 Translation Engine

Translation engine cannot handle hardware diversities, such as X86 processor and AMD processor, or rtl8139 NIC and e1000 NIC. Therefore, we pre-configure QEMU and Xen device emulation module to emulate the same type of devices for Guest OS, such as X86 processor, rtl8139 NIC and etc. We also configure them to have the same amount of memory and to share the same image file as virtual disk. As a result, translation engine only needs to deal with the emulation implementation incompatibility of the same device. For instance, considering the emulation differences of IOAPIC (I/O Advanced Programmable Interrupt Controller) between Xen-HVM and Qemu, we observe that the significant differences are the number of IOAPIC pins and the definition of each redirectory entry. In order to eliminate such kind of device emulation incompatibility, we refer to the Intel IOAPCI datasheet [1] for the functionality of each specific pin, and match them at the granularity of Xen and Qemu device emulation code. Note that only the devices emulated by Xen HVM itself require this kind of scaling and this work needs to be done only once before our system is

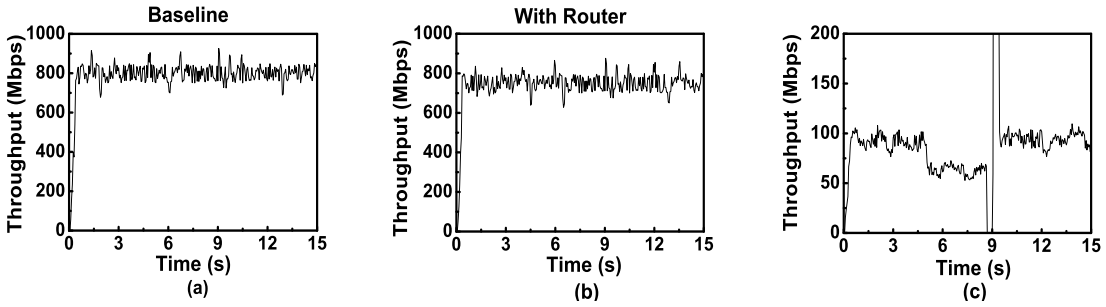


Figure 4: Runtime Overhead and Performance Degradation in terms of Throughput

deployed.

4.3 Infection Analyzer and Reconstruction Engine

As a binary-translation based emulator, Qemu enables our implementation of instruction flow taint analysis. Each executed instruction is audited before Qemu translation block works to keep consistent with the view of Guest OS. Each register, memory cell and disk block are associated with one specific taint bit indicating whether this storage unit is tainted or not. Specifically, when instructions are executed, we apply tainting and untainting policies to examine, set or clear the taint bit. Therefore, we can exactly locate the taint propagation by checking the taint bit throughout the registers, memory cells and disk blocks.

We also implement the reconstruction engine to provide OS semantics. For instance, to dynamically reconstruct process lists, we start from the structure *env* defined by Qemu to emulate the processor for VM. We can obtain all the registers there including the register *esp* pointing to the kernel stack of the currently running process. At the bottom of the kernel stack resides the *thread_info* structure that includes a pointer to the *task_struct* of the corresponding process. Similarly, with the help of other registers and the process profiles, we can further identify the process list, all the file objects and part of kernel data structures (almost 800) such as system call table, interrupt table and etc.

5. EVALUATION

In this section, we evaluate PEDA system in terms of logging efficiency and intrusion analysis comprehensiveness. The Xen hypervisor and the backend system are running separately on two Lenovo Thinkstation Tower D10 machines, each with Dual Intel(R) PRO/1000 NICs. We run CentOS 5.2 (kernel version 2.6.18) as coordination platform on backend system and as Xen Domain 0 on Xen hypervisor. Both Xen-HVM Domain U and Qemu Guest OS are installed with Fedora 6 (kernel version 2.6.20). Both Qemu and Xen-HVM are preconfigured to have the same amount of memory (1 GB), the same NAT based networking, and the same kind of devices emulation for Guest OS or Domain U. We also deploy a router, connecting these two machines with Gigabit Ethernet and responsible for packets direction to them. Outside the router, we have two Dell OptiPlex 745 machines used as client request simulation and attacker platforms, with Gigabit Ethernet connection to the router.

5.1 Logging Efficiency

We install *Apache 1.3.9* on Xen-HVM Domain U as a *http* server. We evaluate the runtime overhead introduced by non-deterministic events plus system call logging, and measure the server performance degradation during the pre-checkpointing phase and service downtime during the stop-and-copy phase.

Runtime Overhead We compare the performances of *apache* running on the native Xen-HVM Domain U and on the Xen-HVM Domain U with our instrumentation. On the Dell machines, we simulate clients sending continuous requests over concurrent connections to fetch an 8 KB file. Figure 4(a) shows that the native Xen-HVM Domain U achieves the throughput of almost 800 Mbps, which is considered as baseline performance in our evaluation. It did not reach up to 2000 Mbps (Two Gigabit NICs on the machine) probably due to the impact of network I/O virtualization introduced by Xen. The baseline performance can be improved by optimizations proposed in [12] to achieve the near-native throughput. Figure 4(b) shows the *apache* throughput with the packets redirected to the logging backend system (from the edge router). Compared with Figure 4(a), we can see that our logging achieves about 95% baseline performance, which the 5% runtime overhead is mainly caused by the system call logging.

Downtime and Performance Degradation Caused by Checkpointing In order to simulate the Amazon-style server during 2 am-5 am, we reduce user requests to *apache* server by 90%. Figure 4(c) shows that the server throughput decreases correspondingly. At the time (about 5 seconds from the very beginning) when we issued the command *xm checkpoint*, the server throughput drops by 24%, which lasts for almost 4 seconds. This performance degradation can be explained by the fact that we introduced a pre-checkpointing phase, during which the whole virtual disk and physical memory are recorded. Following is the service downtime (no throughput) during the "stop-and-copy" phase, which only lasts for less than 0.4 second. To take a checkpoint of a running system, the service downtime cannot be eliminated, because it is generally impossible to take a consistent whole system checkpoint considering the fact that a running system may do "write" operations to either memory or disk. We reduce this kind of service downtime by introducing a pre-checkpointing phase, which takes the large amount of copying workload from the "pausing" phase. Note that in Figure 4(c), the short pulse immediately after the downtime is likely to be caused by the accumulated requests from the performance degradation period.

5.2 Intrusion Analysis Comprehensiveness

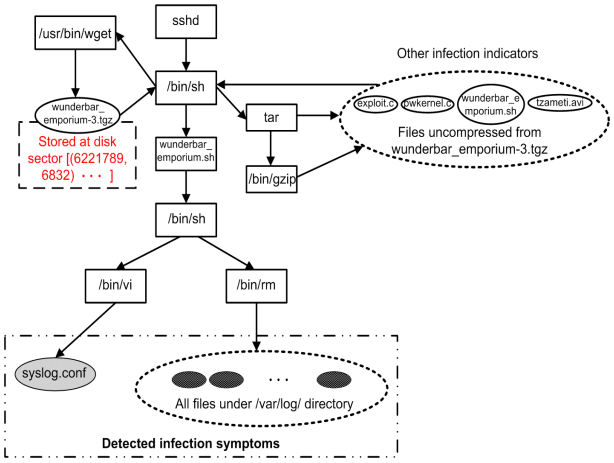


Figure 5: Fine-Grained Intrusion Root Identification

To show the comprehensiveness and precision of our intrusion analysis, we conduct two case studies of real-life intrusion. For the page limit, we only show the detailed results of the first case in terms of fine-grained intrusion root identification and dynamic taint tracking. For the second case, we focus on showing the advance of PEDA system over previous system-call-level intrusion analysis.

5.2.1 Case Study 1

The attack scenario of Case 1 is as follows. The attacker first logs into the server system by *ssh* using an unprivileged user account. Then he downloads a Linux NULL pointer dereference exploit and launches the attack [2] to gain root privilege. Afterwards, he mails back, examines and modifies the *syslog.conf* file to let the system logs be sent to his email account. Finally, he deletes all files under the */var/log/* directory to hide his intrusion "footprint".

Fine-grained Intrusion Root Identification We assume that the IDS detects the maliciously modified file *syslog.conf* and the missing files under the directory */var/log/*. These intrusion symptoms are notified to the PEDA system. Afterwards, we start the intrusion root identification from the detected intrusion symptoms, and trace the automatically-generated dependency graph backward. We tailor the intrusion flows at the system object level from the dependency graph, and locate the system-object-level intrusion root *wget*. Figure 5 shows the fine-grained intrusion root identification procedure. We audit system calls issued by *wget* to identify the buffers containing the intrusion packet. Finally, we can obtain the disk sectors used to store the intrusion packet, which is taken as fine-grained taint seed for infection diagnosis.

Infection Diagnosis In order to show sufficient information regarding the intrusion behaviour, and the details of how the intrusion happens on the server system and what has been infected by the intrusion propagation, we start a whole system dynamic taint tracking from the disk sectors containing the intrusion packet during replay. For the space limitation, Figure 6 presents only partial outcome of our infection diagnosis. The rectangle denotes memory address space or disk sectors on the server system. The ellipse attached to each rectangle includes the OS semantics from our reconstruction engine, including the processes which the address space belongs to, the files which the disk sectors are allocated to, or the dynamic libraries which the memory address space is loaded to. It is sufficient to demonstrate that our cross-layer infection diagnosis features with specific infected memory space, disk sectors, and kernel address space, which are far beyond the system-call-level intrusion tracking. Moreover, the infected memory address information provides the system admin a feasible way to "sweep out" any intrusion harm on the victim system. In addition, we can catch the "blind spot" of system-call-level intrusion analysis. For instance, we can capture how the attacker obtains the root privilege through the intrusion packet.

5.2.2 Case Study 2

Case Study 2 is designed to demonstrate the advance of PEDA system over other system-call-level intrusion analysis. The attacker logs into the system by *ssh* using an unprivileged user account. Then he launches the *sendmail* local escalation exploit to gain root access. The attacker uses the root shell to download and install the *adore* rootkit, which replaces several kernel hooks in the system call table with its own implementation. Afterwards, he uses the same root shell to download and install the *ARK* rootkit, which replaces system binaries (e.g., *syslogd*, *login*, *sshd*, *ls*, *ps*, *netstat*, and etc.) with backdoored versions. We rely on IDS to detect the modification of system binaries by integrity check. By virtue of the backward system call dependency tracking, all of PEDA, *SHELF* [19] and *Backtracking* [11] can identify *ssh* as the system-object-level intrusion root. However, neither *SHELF* nor *Backtracking* can locate the malicious kernel hook modification by *adore* rootkit. Instead, they are only capable of diagnosing the intrusion infection of *ARK* rootkit due to their system call flow auditing. Rather, the fine-grained intrusion root identification of PEDA system can audit the system calls issued by *ssh*, and identify the disk sectors containing the downloaded rootkits *adore* and *ARK* as taint seed. By applying dynamic taint tracking and semantics reconstruction, PEDA is able to capture not only the damage identified by *SHELF* and *Backtracking*, but also the intrusion harm of the kernel hooks modification implanted by *adore* rootkit, such as the replaced *sys_write* and etc.

6. LIMITATIONS

In this section, we discuss the limitations of PEDA system. First, the automatic intrusion backtracking is not 100 % accurate, especially at the granularity of memory cell or disk sector. Our PEDA system relies on intrusion backtracking to locate the fine-grained intrusion root, which in turn is provided as taint seed to infection analyzer. To reduce the false positive on PEDA's intrusion harm analysis result, the intrusion backtracking of PEDA involves some human interference to accurately locate the fine-grained intrusion root. Second, to replay the execution of a busy server with significantly high workload, the amount of non-deterministic events to be recorded might be huge. In this case, it may not be feasible for PEDA to store a history of events that is much longer than the expected intrusion detection delay. Thus, if the intrusion is detected much later than its occurrence, the first run compromised execution cannot be completely replayed due to the removal of long time ago non-deterministic event logs.

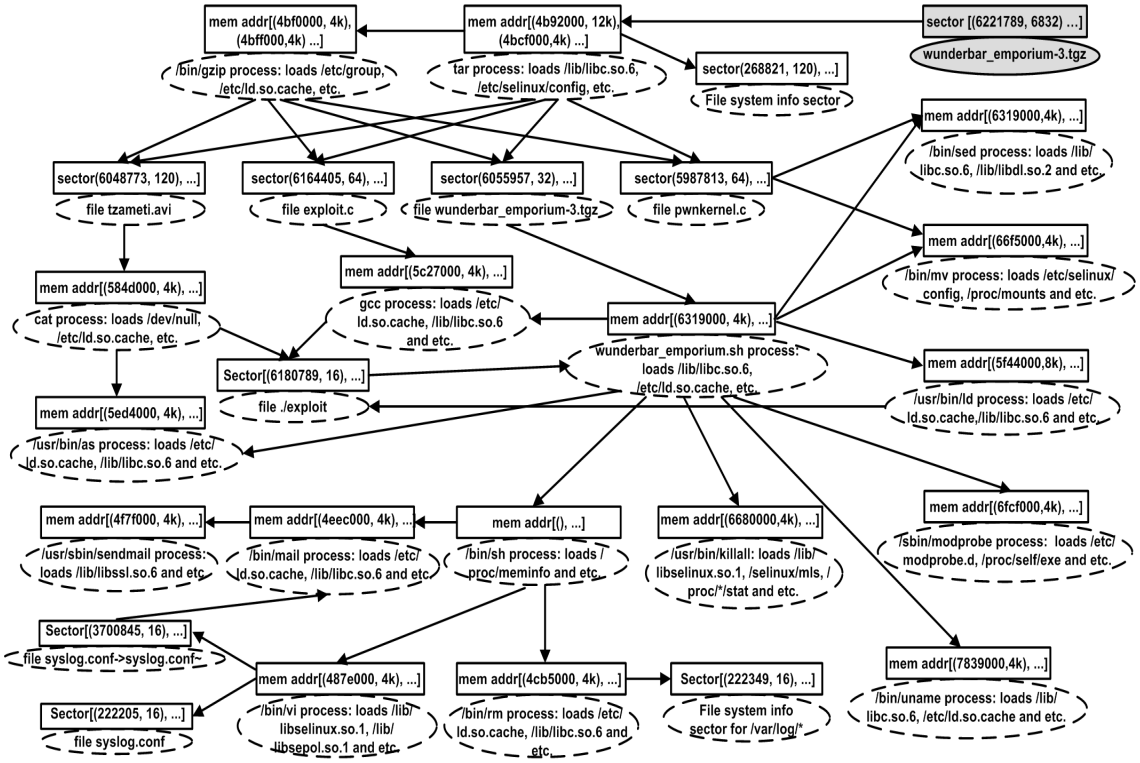


Figure 6: Whole System Infection Diagnosis

7. RELATED WORK

First, our fine-grained intrusion root identification is implemented by integrating backward system-object dependency tracking [11] and forward instruction flow taint analysis. This is the first attempt to bridge the gap between the forward fine-grained analysis and the backward system-object intrusion root identification. As a general intrusion analysis tool, PEDA advances existing system-object-level analysis in terms of intrusion harm comprehensiveness and precision, as shown in our second case study of evaluation section. VM replay is a relatively mature technique in the VM industry (e.g., VMWare) these days. However, the replay on another heterogeneous VM is not. Some new issues exit, such as how to address the device emulation incompatibilities. Aftersight [6] is the first work that we can find talking about these heterogeneous VM migration issues. The following we will mainly discuss the differences between our work and Aftersight.

Being a generic technology for decoupling dynamic program analysis from execution, Aftersight decouples instruction level analyses from the normal execution (of online servers) for a spectrum of purposes, including bug finding and forensics. Aftersight records program execution and replays it on a separate analysis platform against a set of memory safety guarantee policies. Hence, it enables heavyweight analysis during replay to find serious bugs in large complex systems such as VMWare ESX Server and Linux. In contrast, PEDA focuses on the post-mortem intrusion analysis for production workload servers from intrusion root identification to fine-grained infection diagnosis. Thus, the problems faced by PEDA are to precisely locate the intrusion root object to patch the vulnerabilities, and to reasonably associate the intrusion root object with the fine-grained taint seed to start comprehensive infection diagnosis. Though Aftersight and PEDA share the same idea of decoupling analysis from normal execution, they aim at different types of analysis, thus dealing with different sets of design and implementation issues.

In addition to dealing with different analysis, PEDA also differs from Aftersight in the architecture design. Aftersight migrates guest server system from recording platform (VMWare Workstation) to analysis platform (Qemu), while PEDA does it from Xen to Qemu. Since Xen relies on qemu-dm to emulate majority of devices, PEDA takes the approach of recording the external inputs to each device and redelivering them to the corresponding device during the replay. Because VMWare and Qemu emulate I/O devices differently, Aftersight chooses to record all the outputs from each emulated device to CPU and to redeliver them to CPU during the replay to "bypass" the device emulation incompatibility. In order to avoid the significant runtime overhead introduced by the large amount of the device output logging, Aftersight adopts the approach of "replay based replay". In particular, Aftersight records external inputs to the device during normal execution, logs the device outputs to CPU during the first replay, and finally replays the second recording for analysis. Compared with Aftersight, the device emulation incompatibility elimination of PEDA is more straightforward and efficient for production workload server systems, though less generic.

Several other works exist to help security administrators to do intrusion analysis, such as *Repairable File Service* [23], *Intrusion Recovery* [22] and *Backtracking* [11]. All of them log system calls during execution, and use them to track

the flow/dependency between system objects. This kind of coarse-grained dependency tracking typically cannot capture the whole "footprint" of intrusion, because the attackers can craft programs with direct memory load/store instructions that can evade the system call level auditing. Rather, PEDA applies the fine-grained instruction flow taint tracking to capture the intrusion propagation with both comprehensiveness and precision. Both Backtracking [11] and PEDA use backward tracking from detected intrusion symptoms to locate the intrusion root. However, Backtracking only identifies the system-object-level intrusion root, typically a process. In order to provide the infection analyzer the fine-grained taint seed, PEDA extends the Backtracking to "dip" further into the memory cells or disk storage segments granularity. Thus, PEDA can effectively integrate the backward system object intrusion root identification and the the forward fine-grained taint analysis.

8. CONCLUSION

PEDA is a systematic approach doing post mortem fine-grained intrusion analysis for production workload servers. It helps security technicians swiftly identify the fine-grained intrusion root "breakin" to the server and precisely pinpoint the infection propagation throughout the server. PEDA effectively decouples the analysis work off the online server execution by novelly integrating the backward system call dependency tracking and forward fine-grained taint analysis. The proposed heterogeneous VM migration significantly reduces the runtime overhead of online server execution. Our evaluation demonstrates PEDA's advance over existing intrusion analysis systems in terms of efficiency and comprehensiveness. We believe that the comprehensive intrusion analysis functionality of our PEDA system should have a profound impact on any system recovery framework.

9. ACKNOWLEDGMENTS

We extend thanks to our shepherd, Reiner Sailer, for valuable feedback and constructive suggestions. We thank all the anonymous reviewers for carefully reading the drafts and providing helpful revision comments. This work was supported by AFOSR FA9550-07-1-0527 (MURI), ARO W911NF-09-1-0525 (MURI), NSF CNS-0905131, and AFRL FA8750-08-C-0137.

10. REFERENCES

[1] Intel i/o apic datasheet. http://www.intel.com/design/chipsets/datashts/290566.htm.

[2] Linux null pointer dereference. http://archives.neohapsis.com/archives/fulldisclosure/2009-08/0174.html.

[3] F. Bellard. Qemu, a fast and portable dynamic translator. *USENIX Annual Technical Conference*, 2005.

[4] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, pages 80–107, 1996.

[5] P. M. Chen and B. D. Noble. When virtual is better than real. *HotOS*, 2001.

[6] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. *USENIX Annual Technical Conference*, 2008.

[7] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P.Barham. Vigilante end-to-end containment of internet worms. *SOSP*, 2005.

[8] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *Proceedings of the 5th OSDI*, pages 211–224, 2002.

[9] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. *Eurosys*, 2006.

[10] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. *Proceedings of the 14th ACM CCS*, pages 128–138, 2007.

[11] S. T. King and P. M. Chen. Backtracking intrusions. *SOSP*, 2003.

[12] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing network virtualization in xen. *USENIX Annual Technical Conference*, 2006.

[13] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. *EUROSYS*, 2006.

[14] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, pages 391–411, 1997.

[15] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. *USENIX Annual Technical Conference*, 2005.

[16] A. Slowinska and H. Bos. Pointless tainting?: Evaluating the practicality of pointer tainting. *Eurosys*, 2009.

[17] G. E. Suh, J. W. Lee, D. Zhang, and S. Devada. Secure program execution via dynamic information flow tracking. *ASPLOS*, 2004.

[18] Z. Wang, X. Jiang, W. Cui, and X. Wang. Countering persistent kernel rootkits through systematic hook discovery. *RAID*, 2008.

[19] X. Xiong, X. Jia, and P. Liu. Shelf: Preserving business continuity and availability in an intrusion recovery system. *ACSAC*, 2009.

[20] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. *MoBS*, 2007.

[21] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. *CCS*, 2007.

[22] S. Zhang, X. Xiong, X. Jia, and P. Liu. Availability-sensitive intrusion recovery. *VMSec*, 2009.

[23] N. Zhu and T. Chiueh. Design, implementation, and evaluation of repairable file service. *DSN*, 2003.

Forenscope: A Framework for Live Forensics

Ellick Chan† emchan@illinois.edu
Shivaram Venkataraman† venkata4@illinois.edu
Francis David* francis.david@microsoft.com
Amey Chaugule† achaugu2@illinois.edu
Roy Campbell† rhc@illinois.edu

ABSTRACT

Current post-mortem cyber-forensic techniques may cause significant disruption to the evidence gathering process by breaking active network connections and unmounting encrypted disks. Although newer live forensic analysis tools can preserve active state, they may taint evidence by leaving footprints in memory. To help address these concerns we present Forenscope, a framework that allows an investigator to examine the state of an active system without the effects of taint or forensic blurriness caused by analyzing a running system. We show how Forenscope can fit into accepted workflows to improve the evidence gathering process.

Forenscope preserves the state of the running system and allows running processes, open files, encrypted filesystems and open network sockets to persist during the analysis process. Forenscope has been tested on live systems to show that it does not operationally disrupt critical processes and that it can perform an analysis in less than 15 seconds while using only 125 KB of memory. We show that Forenscope can detect stealth rootkits, neutralize threats and expedite the investigation process by finding evidence in memory.

Keywords: forensics, introspection, memory remanence

1. INTRODUCTION

Current forensic tools are limited by their inability to preserve the hardware and software state of a system during investigation. Post-mortem analysis tools require the investigator to shut down the machine to inspect the contents of the disk and identify artifacts of interest. This process breaks network connections and unmounts encrypted disks causing significant loss of potential evidence and possible disruption of critical systems. In contrast, live forensic tools can allow an investigator to inspect the state of a running machine without disruption. However existing tools can overwrite evidence present in memory or alter the contents of the disk causing forensic *taint* which lowers the *integrity* of the evidence. Furthermore, taking a snapshot of the system can result in a phenomena known as forensic *blurriness* [26] where an inconsistent snapshot is captured because the system is running while it is being observed. Forensic blurriness affects the *fidelity* and quantity of evidence acquired and can cast doubt on the validity of the analysis, making the courts more reluctant to accept such evidence [4].

Experts at the SANS institute and DOJ are starting to recognize the importance of volatile memory as a source of evidence to help combat cybercrime [1, 3]. In response, the SANS institute recently published a report on volatile memory analysis [7]. To help address the limitations of existing volatile memory analysis tools we present Forenscope, a framework for live forensics, that can capture, analyze and explore the state of a computer without disrupting the system or tainting important evidence. Section 2 shows how Forenscope can fit into accepted workflows to enhance the evidence gathering process.

Forenscope leverages DRAM memory remanence to preserve the running operating system across a "state-preserving reboot"(Section 3) which recovers the existing OS without having to go through the full boot-up process. This process enables Forenscope to gain complete control over the system and perform taint-free forensic analysis using well grounded introspection techniques [22]. Finally, Forenscope resumes the existing OS, preserving active network connections and disk encryption sessions causing minimal service interruption in the process. Forenscope captures the contents of system memory to a removable USB device and activates a software write blocker to inhibit modifications to the disk. To maintain fidelity, it operates exclusively in 125 KB of unused legacy conventional memory and does not taint the contents of extended memory. Since Forenscope preserves the state of a running machine, it is suitable for use in production and critical infrastructure environments. We have thoroughly tested and evaluated Forenscope on an SEL-1102, a power substation industrial computer, and an IBM desktop workstation. The machines were able to perform their duties under a variety of test conditions with minimal interruption and running Forenscope did not cause any network applications to time out or fail. Our current implementation is based on Linux 2.6, although the technique is also applicable to other major operating systems.

We have implemented several modules that can check for the presence of malware, detect open network sockets and locate evidence in memory such as rootkit modifications to help the investigator identify suspicious activity.

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

†University of Illinois, *Microsoft

The contributions of this work include:

1. An extensible software framework for high-fidelity live forensics conforming to the best practices of a legal framework of evidence.
2. Efficient techniques to gather, snapshot and explore a system without bringing it down.
3. Implementation and evaluation on several machines including a standard industrial machine and against several anti forensics rootkits.

This paper is organized as follows: Section 2 introduces cyber-forensics followed by Section 3 which describes the design of Forenscope. We evaluate the effectiveness of Forenscope in Section 4. Section 5 discusses forensics issues, Section 6 surveys related work and Section 7 concludes.

2. BACKGROUND

To provide an overview of the accepted legal framework of evidence collection currently in place, we summarize the workflow from the CERT guide on FBI investigation [10]:

1. Preserve the state of the computer by creating a backup copy of logs and files left by the intruder.
2. If the incident is in progress, log activity.
3. Document the losses suffered by your organization.
4. Contact law enforcement.

While the steps executed are similar for various cases, there are special requirements for each case. For instance, in criminal investigation, integrity and fidelity of the data is paramount. As evidence presented in court must be as accurate as possible, special steps must be taken to ensure fidelity. For incident response, the goal is to detect and react to security breaches while minimizing the intrusiveness of the process. In some critical systems it is impractical to interrupt the system to perform forensic analysis of a potential breach and service level agreements (SLAs) may impose financial penalties for downtime. The cases chosen above are example of evidentiary requirements but a more thorough analysis is beyond the scope of this paper. To preserve the fidelity of the original evidence, many forensic workflows capture a pristine image of the evidence and draw conclusions based on analysis of the copy. Conventional post-mortem forensic workflows perform this task by physically shutting down a computer and copying the contents of the hard drive for subsequent analysis. On the other hand, live forensics are often desired for step 2 because they provide access to networked resources such as active SSH and VPN sessions, remote desktop connections, IM clients and file transfers. However even state-of-the-art solutions often cannot image a system with high fidelity and frequently introduce taint in the process. In summary, existing tools require the investigator to make a tradeoff between increased fidelity through post mortem analysis or the potential to collect important volatile information using live forensic tools at the cost of tainting evidence.

One of the key issues in collecting volatile information is that various forms of data such as CPU registers, memory, disk and network connections have different lifetimes. To maximize evidence preservation, RFC 3227 [8] outlines the *order of volatility* of these resources and dictates the order in which evidence should be collected for investigation. Commercial products currently used by forensic experts for incident response such as Encase, Helix, FTK Imager and Memoryze [1] etc, do not capture all forms of data. A comparison of these products is presented in Table 1. Scalpel and Sleuth kit are solely designed for disk analysis while other tools such as Encase, Helix and FTK include some level of memory capture and analysis capability. Memoryze is the only tool listed in the table that performs volatile memory analysis. Some tools such as Helix, FTK and Memoryze can list the state of open network sockets, but the underlying network connections are not preserved during the analysis process. All live forensic tools listed in this table rely on the integrity of the running kernel. Compromised systems may provide inaccurate information. Evidence preservation and minimizing forensic intrusiveness are hard problems that haven't been adequately addressed in the literature.

In contrast, Forenscope was built to comply with steps 1 and 2 where it maximizes the preservation of evidence and avoids disruption of ongoing activities to allow the capture of high fidelity evidence. As a result, we believe that Forenscope may be more broadly applicable to various scenarios which require live forensics such as incident response and criminal investigation. For incident response, we recognize that the integrity of the machine may be violated by malware and our solutions have been designed to address this scenario. For criminal investigation, we presume that the machine may have various security mechanisms implemented such as encrypted disks coupled with authentication mechanisms such as logon screens and screensaver locks.

3. DESIGN

Forenscope utilizes the principle of introspection to provide a consistent analysis environment free of taint and blurriness which we term as the *golden state*. In this state, the system is essentially quiescent and queries can be made to analyze the system. As a result, analysis modules can access in-memory data structures introspectively. The investigator activates forenscope by forcing a reset where the state of the machine is preserved by memory remanence in the DRAM chips. Then, the investigator boots off the Forenscope media which performs forensic analysis on the latent state of the system and restores the functionality of the system for further live analysis. Forenscope is designed to work around security mechanisms by interposing a lightweight analysis platform beneath the operating system. For example, in incident response, the machine may be controlled by malicious software and the operating system cannot be trusted. The observation capabilities afforded by Forenscope offer additional visibility in these scenarios.

3.1 Taint and Blurriness

Taint and blurriness are concepts related to the use of forensic tools. Taint is a measurement of change in the system induced by the use of a forensic tool and it may be present both in memory and on disk. In this section, we only consider the in-memory portion because BitBlocker (Section 3.6) eliminates disk taint by blocking writes. Blurriness refers to the inconsistency of a memory snapshot taken while a system is running.

[1]Encase: www.encase.com,
Helix: www.e-fense.com,
FTK Imager: www.accessdata.com,
Memoryze: www.mandiant.com/software/memoryze.htm

Table 1: Comparison of Forenscope with existing forensic tools

Evidence	Registers	Memory	Network	Processes	Disk	Encryption
RFC 3227 Reqs	Nanosecs	Seconds	Minutes	Minutes	Hours	Hours
Encase	×	✓[a]	×	×	✓	×
Helix	×	✓[a]	✓[b]	✓	✓	×
FTK	×	✓[a]	✓	✓	✓	✓
Scalpel	×	×	×	×	✓	×
Memoryze	×	✓[a]	✓[b]	✓	×	×
Sleuth kit	×	×	×	×	✓	×
Forenscope	✓	✓	✓	✓	✓	✓

[a] Subject to forensic blurriness
[b] Connection is recorded but not persisted

Table 2: Definitions

Quantity	Description
Snapshot $\overline{S_t}$	Contents of memory at time t
Natural drift δ_v	Change in the system state over time v
Snapshot $\hat{S_v}$	Contents of captured memory snapshot with v being the time taken to capture the snapshot
Taint f	f is defined as the memory taint caused by the forensic introspection agent

To quantify the relationship between taint and blurriness, let $\overline{S_t}$ be the contents of memory at any given instant of time t. The state of a system changes over a period of time due to the natural course of running processes and we define this as the natural drift of the system, δ. When a traditional live forensic tool attempts to take a snapshot of the system, there is a difference between what is captured, $\hat{S_v}$ and the true snapshot $\overline{S_t}$, where v represents the time taken to capture the snapshot. There are two reasons for this difference: the first being δ_v the natural drift over the time period when the snapshot was being acquired (v) and the second due to the footprint f of the forensic tool. We define the former as the blurriness of the snapshot and the latter quantity to be the taint caused by the forensic tool. Table 2 captures these definitions in a concise form. In general, there are two ways to obtain a snapshot of the machine's state: active techniques and passive techniques. Active techniques involve the use of an agent on the machine which may leave a footprint. Passive techniques operate outside the domain of the machine and do not affect its operation, one such example is VM introspection. When a passive acquisition tool is used, the relationship $\hat{S_v} = \overline{S_t} + \delta_v$ indicates that the approximate snapshot differs from the true snapshot due to the blurriness δ_v. In contrast, when an active forensic tool is used, $\hat{S_v} = \overline{S_t} + f + \delta_v$, where f represents taint and δ_v represents blurriness. Collectively, these quantities are a measure of the error in the snapshot acquisition process. Taint can result from the direct action of forensic tools or indirect effects induced in the system through the use of these tools. We call the former first-order taint, f', and the latter second-order taint, f''. First-order taint can result from loading a forensic tool into memory and second-order taint can result from processes such as file buffering due to the effects of a forensic tool writing a file.

3.2 Memory Remanence

Modern memory chips are composed of capacitors which store binary values using charge states. Over time, these capacitors leak charge and must be refreshed periodically. To save power, these chips are designed to retain their values as long as possible, especially in mobile devices such as laptops and cell phones. Contrary to common belief, the act of rebooting or shutting down a computer often does not completely clear the contents of memory. Link and May [21] were the first to show that current memory technology exhibited remanence properties back in 1979. More recently, Gutmann [18] elaborated on the properties of DRAM memory remanence. Halderman et al. [19] recently showed that these chips can retain their contents for tens of seconds at room temperature and the contents can persist for several minutes when the RAM chips are cooled to slow the natural rate of bit decay. Forenscope utilizes memory remanence properties to preserve the full system state to allow recovery to a point where introspection can be performed. We refer the reader to [11, 19] for a more detailed analysis of memory remanence.

3.3 Activation

Forenscope currently supports two methods of activation. The first is based on a watchdog timer reset and the second is through a forced reboot. For incident response, a watchdog timer may be used to activate Forenscope periodically to audit the machine's state and check for the presence of stealth malware. Watchdog timers are used in embedded systems to detect erroneous conditions such as machine lockups. These timers contain a count down clock which must be refreshed periodically. If the system crashes, the watchdog software will fail to refresh the clock. Once the clock counts down to zero, the watchdog timer will issue a warm hardware reset signal to the machine causing it to reboot in the hopes that the operating system will recover from the erroneous condition upon a fresh start. On our test machine, the built-in watchdog timer is programmable via a serial port interface and the contents of DRAM memory are not cleared after a reboot initiated by the watchdog timer reset signal.

On the other hand, a forensic investigator may encounter a machine that is locked by a screensaver or login screen and in this situation, Forenscope can be activated by forcing a reboot. Some operating systems such as Linux and Windows can be configured to reboot or produce a crash dump by pressing a hotkey. These key sequences are often used for

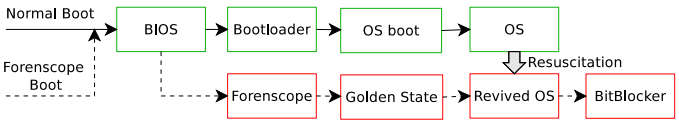


Figure 1: Forenscope vs normal boot paths

debugging and are enabled by default in many Linux distributions. In Linux, the *alt-sysrq-b* hotkey sequence forces an immediate reboot. If these debug keys are disabled, then a reset may be forced by activating the hardware reset switch. Forenscope supports multiple modes of operation for versatility. After the machine has been rebooted forcefully, the Forenscope kernel is selected from the boot loader menu instead of the incumbent operating system.

3.4 Forenscope framework

Instead of booting afresh, Forenscope alters the boot control flow to perform its analysis. Figure 1 illustrates this process. After the machine restarts, it boots off a CD or USB stick with the Forenscope media. The machine then enters the *golden state* monitor mode which suspends execution and provides a clean external view of the machine state. To explain how the monitor works, we first describe the operating states of the x86 architecture. When a traditional PC boots, the processor starts in *real mode* and executes the BIOS. The BIOS then loads the bootloader which in turn loads the operating system. During the boot sequence, the operating system first enables `protected mode` to access memory above the 1 MB mark and then sets up page tables to enable virtual memory to bootstrap the OS. Forenscope interposes on this boot sequence and first establishes a bootstrap environment residing in the lower 640 KB rung of legacy conventional memory and then it reconstructs the state of the running machine. Forenscope has full control of the machine and its view is untainted by any configuration settings from the incumbent operating system because it uses a trustworthy private set of page tables; thus rootkits and malware which have infected the machine cannot interfere with operations in this state. Next, Forenscope obtains forensically-accurate memory dumps of the system and runs various kinds of analyses. For integrity, Forenscope does not rely on any services from the underlying operating system. Instead, it makes direct calls to the system's BIOS to read and write to the disk. Therefore, Forenscope is resistant to malware that impedes the correct operation of hardware devices. The initial forensic analysis modules are executed in this state and then Forenscope restores the operation of the incumbent operating system.

3.5 Reviving the Operating system

To revive the incumbent operating system, Forenscope needs to restore the hardware and software state of the system to "undo" the effects of the reboot. Hardware devices are reset by the BIOS as part of the boot process. Some of these devices must be reconfigured before the incumbent operating system is restored because they were used by Forenscope or the BIOS during initialization. To do so, Forenscope first re-initializes core devices such as the hard drive and interrupt controller and then assumes full control of these devices for operation in its clean environment. Before resuming the operating system, Forenscope scans the PCI bus and gathers a list of hardware devices. Each hardware device is matched against an internal database and if an entry is found, Forenscope calls its own reinitialization function for the particular hardware device. If no reinitialization function is found, Forenscope looks up the device class and calls the operating system's generic recovery function for that device class. Many devices such as network cards and disk drives have facilities for handling errant conditions on buggy hardware. These devices typically have a *timeout recovery* function which can revive the hardware device in the event that it stops responding. We have found that calling these recovery functions is usually sufficient to recover most hardware devices. In Linux, 86 out of the 121 (71%) PCI network drivers implement this interface and all IDE device drivers support a complete device reset. For instance, the IBM uses an Intel Pro/100 card and the SEL-1102 uses a built-in AMD PCnet/32 chip. On both these machines Forenscope relies on calling the `tx_timeout` function to revive the network. We use a two-stage process to restore the operating system environment. The first stage reconstructs the processor state where the values of registers are extracted and altered to roll back the effects of the restart and the second stage runs forensic analysis modules. Our algorithm scans the active kernel stack and symbol information from the kernel for call chain information. Forenscope uses this information to reconstruct the processor's state. In the `alt-sysrq-b` case, the interrupt handler calls the keyboard handler which in turn invokes the emergency sysrq-handler. The processor's register state is saved on the stack and restored by using state recovery algorithms from [11, 13]. If the `alt-sysrq-b` hotkey is disabled, Forenscope supports an alternate method of activation based on pressing a physical reset switch. In this case, Forenscope assumes that the system is under light load and that the processor spends most of its time in the kernel's idle loop. In this loop, most kernels repeatedly call the x86 `HLT` instruction to put the processor to sleep. Since the register values at this point are predictable, Forenscope restores the instruction pointer, `EIP`, to point to the idle loop itself and other registers accordingly. Once the state has been reconstructed, Forenscope reloads the processor with this information and enables virtual memory.

3.6 Modules

We have developed a number of modules to aid in forensic analysis. These modules, shown in Figure 2, run in groups where stage 1 modules run in the golden state to collect pristine information while stage 2 modules rely on OS services to provide a shell and block disk writes. Finally, stage 3 resumes the original operating environment.

Scribe: Scribe collects basic investigation information such as the time, date, list of PCI devices, processor serial number and other hardware features. These details are stored as evidence to identify the source of a snapshot.

Cloner: Cloner is a memory dump forensic tool that is able to capture a high-fidelity image of volatile memory contents to an external capture device. Existing techniques for creating physical memory dumps are limited by their reliance on system resources which are vulnerable to deception. Cloner works around forensic blurriness issues and rootkit cloaking by running in stage 1 before control is returned to the original host OS. In the golden state, the system uses `protected mode` to access memory directly through Forenscope's safe memory space. Using this technique, Cloner accesses memory directly without relying on services from the incumbent operating system or its page tables. To dump the contents of memory, Cloner writes to disk directly using BIOS services instead of using an OS disk driver. This channel avoids a potentially booby-trapped or corrupted operating system disk driver and ensures that the written data has better forensic integrity. Most BIOS firmware supports read/write access

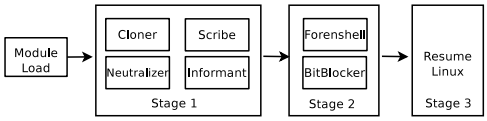


Figure 2: Forenscope modules

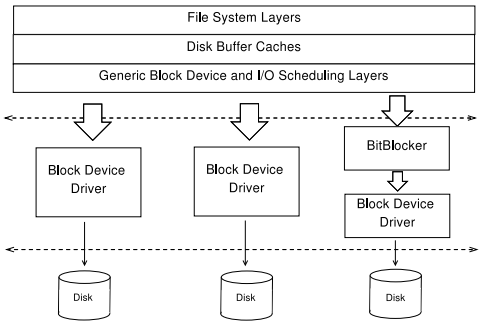


Figure 3: File system architecture

to USB flash drives and hard disks. Another reason to use the BIOS for dumping is that it minimizes the memory footprint of Forenscope and reduces dependencies on drivers for various USB and SATA chipsets. Once cloner captures a clean memory dump, the investigator can run other modules tools that may alter the contents of memory without worry of tainting the evidence.

Informant: Informant checks for suspicious signs in the system that may indicate tampering by identifying the presence of alterations caused by malware. In order to extract clean copies of the program code and static structures such as the system call table, Forenscope must have access to a copy of the `vmlinux` kernel file which is scanned to locate global kernel variables and the location of various functions. Most Linux distributions provide this information. Read-only program code and data structures are checked against this information to ensure that they have not been altered or misconfigured. Such alterations have the potential to hinder the investigation process and Informant helps to assess the integrity of a machine before further analysis is attempted. After Informant verifies the system, it also records other useful information such as the contents of the kernel *dmesg* log, running processes, open files and open network sockets. This information can help expedite the investigation process.

Neutralizer: Neutralizer inoculates against anti-forensic software by detecting and repairing alterations in binary code and key system data structures such as the system call table. These structures can be repaired by restoring them with clean copies extracted from the original sources. Since many rootkits rely on alteration techniques, Neutralizer can recover from the effects of common forms of corruption. Presently, Neutralizer is unable to recover from corruption or alteration of dynamic data structures. Neutralizer also suppresses certain security services such as the screensaver, keyboard lock and potential malware or anti-forensic tools by terminating them. To terminate processes, neutralizer sends a `SIGKILL` signal instead of a `SIGTERM` signal so that there is no opportunity to ignore the signal. Customized signals can be sent to each target process. For some system services that respawn, terminating them is ineffective, so forcefully changing the process state to zombie (Z) or uninterruptible disk sleep (D) is desired instead of killing the application directly. An alternative would be to send the `SIGSEGV` signal to certain applications to mimic the effects of a crash. Neutralizer selects processes to kill based on the analysis mode. For incident response on server machines, a white list approach is used to terminate processes that do not belong to the set of core services. This policy prevents running unauthorized applications that may cause harm to the system. For investigation, Neutralizer takes a black list approach and kills off known malicious processes.

Table 3: Correctness assessment

Application	Results
Idle system	System is correctly recovered over 100 times.
SSH	SSH recovers, protocol handles lost packets.
PPTP VPN	VPN recovers, queued messages are delivered.
AES pipe	File encryption continues.
Netcat	File transfers correctly without checksum errors.
DM-crypt	Mounted filesystem remains accessible.

ForenShell: ForenShell is a special superuser *bash* shell that allows interactive exploration of a system by using standard tools. When coupled with BitBlocker(below), ForenShell provides a safe environment to perform customized live analyses. In this mode, Forenshell becomes non-persistent and it does not taint the contents of storage devices. Once ForenShell is started, traditional tools such as Tripwire or Encase may be run directly for further analysis. To provide an audit log of the investigator's activities, ForenShell provides a built-in keylogger that writes directly to the evidence collection medium without tainting the disk. Forenscope launches the superuser shell on a virtual console by directly spawning it from a privileged kernel thread. ForenShell runs as the last analysis module after Informant and Neutralizer have been executed. At this point, the system has already been scanned for malware and anti-forensic software. If Neutralizer is unable to clean an infection, it displays a message informing the investigator that the output of ForenShell may be unreliable due to possible system corruption.

BitBlocker: BitBlocker is a configurable software-based write blocker that inhibits writing to a given set of storage devices to avoid tainting the contents of persistent media. Since actions performed by ForenShell during exploration can inadvertently leave undesired tracks, BitBlocker helps to provide a safe non-persistent analysis environment that emulates disk writes without physically altering the contents of the media. Because BitBlocker modifies the contents of memory, it executes after Cloner has captured a clean copy of memory.

Simply re-mounting a disk in read-only mode to prevent writing may cause some applications to fail because they may need to create temporary files and expect open files to remain writable. Typically, when an application creates or writes files, the changes are not immediately flushed to disk and they are held in the disk's buffer cache until the system can flush the changes. The buffer cache manages intermediate disk operations and services subsequent read requests with pending writes from the disk buffer when possible. BitBlocker mimics the expected file semantics of the original system by reconfiguring the kernel's disk buffer cache layer to hold all writes instead of flushing them to disk. This approach works on any type of file system because it operates directly on the disk buffer which is one layer below the file system. BitBlocker's design is similar to that of some Linux-based RAM disk systems [5] which cleverly use the disk buffer as a storage system by configuring the storage device with a null backing store instead of using a physical disk. Each time a disk write is issued, barring a *sync* opera-

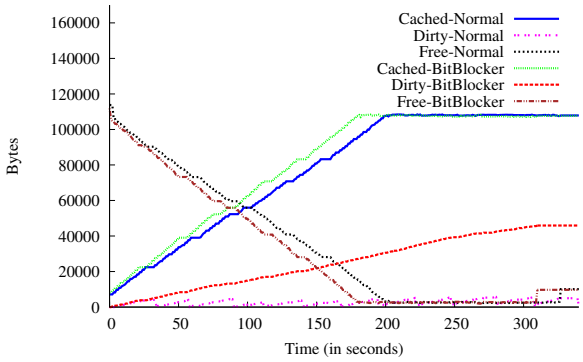


Figure 4: BitBlocker memory usage

tion, the operating system's disk buffer subsystem holds the request in the buffer until a certain write threshold or timeout is reached. In Linux, a system daemon called *pdflush* handles flushing buffered writes to disk. To prevent flushing to the disk, BitBlocker reconfigures the write threshold of the disk to inhibit buffer flushing, disables *pdflush* and hooks the *sync, sync_file_range, fsync, bdflush* and *umount* system calls with a write monitor wrapper. Figure 3 shows the architectural diagram of the Linux filesystem layer and where BitBlocker intercepts disk write operations. Although BitBlocker inserts hooks into the operating system, it does not interfere with the operations of Informant and Neutralizer because those modules are run before BitBlocker and they operate on a clean copy of memory. The hooks and techniques used by BitBlocker are common to Linux 2.6.x kernels and they are robust to changes in the kernel version. Similar techniques are possible for other operating systems.

4. RESULTS AND EVALUATION

We evaluate Forenscope as a forensic tool by measuring five characteristics: correctness, performance, downtime, fidelity and effectiveness against malware.

Hardware and Software Setup: To demonstrate functionality, we tested and evaluated the performance of Forenscope on two machines: a Schweitzer 1102 industrial computer and an IBM Intellistation M Pro. The SEL-1102 used in our experiments is a rugged computer designed for power system substation use and it is equipped with 512 MB of DRAM and a 4 GB compact flash card mounted in the first drive slot as the system disk. The SEL-1102 can operate in temperatures ranging from -40 to +75 degrees Celsius. The IBM Intellistation M Pro is a standard desktop workstation equipped with 1 GB of DRAM. For some tests, we opted to use a QEMU-based virtual machine system to precisely measure timing and taint. Forenscope and the modules that we developed were tested on the Linux 2.6 kernel. Although Forenscope was originally built to target Linux, we plan to expand this work to other systems.

Correctness: To show that Forenscope is robust, we tested it against a collection of applications listed in Table 3. In each case, after rebooting the machine forcefully, Forenscope recovered the operating state, took control and ran successfully without breaking the semantics of the application. As a basic sanity test, Forenscope was able to revive an idle system with no load. We chose a mix of applications to show that a wide range of hardware, software and network applications are compatible. Running SSH, PPTP and Netcat showed that network connections persist. Further

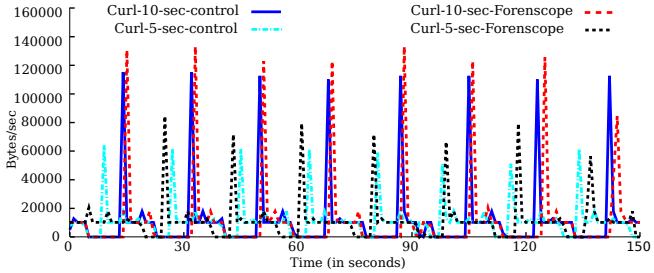


Figure 5: HTTP data transfer rate comparison

testing using DM-crypt and AES pipe showed that security programs continue to operate properly. A more thorough evaluation of the correctness can be found in [11]. To evaluate the correctness of BitBlocker, we ran it on the IBM and on a QEMU system emulator. Using the emulator allowed us to verify integrity by checksumming the contents of the virtual disk. Our test cases include using the *dd* utility to fill up the disk, then issuing a *sync* command and unmounting the disk. Other cases tested include copying large files and compiling programs consisting of hundreds of files. In each case, BitBlocker worked correctly and no writes were issued to the physical disk. After the test completed, we confirmed that the contents of the disk were unchanged by comparing hashes of the contents against the original contents.

Performance: In terms of performance, BitBlocker made disk operations appear to be faster because no data is flushed to the physical disk from the disk buffer. A write of a 128 MB file took 32.78 s without BitBlocker and 3.71 s with BitBlocker. The number of dirty disk buffers consumed increases proportionately with the size of the files written. Since BitBlocker inhibits flushing to disk, running out of file buffers can create a condition where the filesystem fills up and reports a write error. To measure these effects on the system, we collected buffer cache usage information once a second in several key applications: creating a compressed archive with *tar-bzip2*, downloading a file using *wget* and compiling the software package *busybox*. Figure 4 shows the utilization of dirty file buffers over time for the tar-gzip case. Wget and busybox compilation have similar results. In the graphs, we report statistics from `/proc/meminfo` such as `cached`, `dirty` and `free`. According to the documentation for `/proc`, `cached` in Linux represents the amount of data in the page cache which includes cached data from read-only files as well as write buffers. `Dirty` represents items that need to be committed to the disk and free represents free memory. From our observations, `dirty` is generally very low in the normal case because the kernel commits write buffers periodically. However, in BitBlocker, `dirty` grows steadily because the data cannot be committed back to the disk. To estimate the amount of memory required to run BitBlocker, our experiments show that in many scenarios, even 128 MB of free memory is sufficient for BitBlocker to operate. Our experiments show that BitBlocker is robust even when the system runs low in memory. At 200 seconds, the physical memory of the machine fills up and the tar-bz2 process stops because the disk is "full." The system does not crash and other apps continue to run as long as they do not write to the disk. On a typical system with 2 GB of memory, BitBlocker should be able to maintain disk writeability for a much longer period of time.

Table 4: Taint measurement (pages)

Description (32,768)	Conventional Memory	Extended Memory
Forenscope	41 (0.125%)	0(0%)
dd	0 (0%)	7100 (21.66%)
dd to FS mounted with sync flag	0 (0%)	7027 (21.44%)
dd with O_DIRECT	0 (0%)	480 (1.46%)

Downtime: As discussed earlier, one important metric for evaluating a forensic tool is the amount of downtime incurred during use. To show that Forenscope minimally disrupts the operation of critical systems, we measured the amount of time required to activate the system. Forenscope, without Cloner, executed in 15.1 s using the reboot method on the SEL-1102 and in 9.8 s on the IBM Intellistation while the watchdog method took 15.2 s to execute on the SEL-1102. The majority of the downtime is due to the BIOS bootup sequence and this downtime can be reduced on some machines. Many network protocols and systems can handle this brief interruption gracefully without causing significant problems. We tested this functionality by verifying that VPN, SSH and web browser sessions continue to work without timing out despite the interruption. Many of these protocols have a timeout tolerance that is sufficiently long to avoid disconnections while Forenscope is operating and TCP is designed to retransmit lost packets during this short interruption. To measure the disruption to network applications caused by running Forenscope continuously over a period of time, we ran a test within a virtualized environment to mimic the brief reboot cycle used by the analysis process. The test measures the instantaneous speed of an HTTP file transfer between a server and a client machine. While the file transfer is in session, we periodically interrupt the transfer by forcibly restarting the machine and subsequently reviving it using Forenscope. Each time the system is interrupted, the server process is suspended while the machine reboots. The process is then resumed once Forenscope is done running. As a baseline, we created a control experiment where the server process is periodically suspended and resumed by a shell script acting as a governor to limit the rate at which the server operates. This script sends the `SIGSTOP` signal to suspend the server process, waits a few seconds to emulate the time required for the bootup process and then sends a `SIGCONT` signal to resume operation. In each experiment, a `curl` client fetches a 1 MB file from a `thttpd` server at a rate of 10 KB/s. We chose these parameters to illustrate how a streaming application or low-bandwidth application such as a logger may behave. During this download process, the server was rebooted once every 20 seconds and we measured the instantaneous bandwidth with a bootup delay of 5 and 10 seconds to observe the effects of various bootup times. We observed that the bandwidth drops to zero while the system boots and the download resumes promptly after the reboot. No TCP connections were broken during the experiment and the checksum of the downloaded file matched that of the original file on the server. A graph of the instantaneous bandwidth vs time is plotted in Figure 5. We compared the results of our test against the control experiment and observed that the behavior was very similar. Thus we believe that running Forenscope can be considered as safe as suspending and resuming the process. During the experiment we noticed that the bandwidth spiked immediately after the machine recovered and attribute this behavior to the internal 2-second periodic timer used by *thttpd* to adjust the rate limiting throttle table.

Taint and Blurriness: We evaluated the taint in a snapshot saved by Forenscope using a snapshot captured by `dd` as the baseline. In an experimental setup running with 128 MB of memory, we collected an accurate snapshot $\overline{S_t}$ of the physical memory using QEMU and compared that with a snapshot $\hat{S_v}$ obtained from each forensic tool. The number of altered pages for each of the configurations is presented in Table 4. We observe that since Forenscope is loaded in conventional memory, the only pages which differ are found in the lower 640 KB of memory. Our experiments show that Forenscope is far better than `dd` because we observed no difference in the extended memory between the snapshot taken by Forenscope and the baseline snapshot. It should be noted that as the machine is suspended in the golden state when running Forenscope, there is no blurriness associated with the snapshot taken by Forenscope. For *dd*, we measured the taint when using a file system mounted with and without the *sync* option. The number of pages affected remains almost the same in both cases and we observed that the majority of second-order taint was due to the operating system filling the page-cache buffer while writing the snapshot. To evaluate how much taint was induced due to buffering, we ran experiments in which *dd* was configured to write directly to disk, skipping any page-cache buffers by using the `O_DIRECT` flag. The results show that the taint was much lower than the earlier experiment, but still greater than the taint caused by using Forenscope. In order to estimate the amount of blurriness caused when tools like *dd* are used, we measured the natural drift over time of some typical configurations. We collected and compared memory dumps from Ubuntu 8.04 and Windows Vista with 512 MB of memory in a virtual machine environment hosted in `QEMU`. In each case, we snapshot the physical memory of the virtual machine and calculate the number of pages that differ from the initial image over a period of time. The snapshots were sampled using a *tilted time frame* to capture the steady state behavior of the system in an attempt to measure δ_v. The samples were taken at 10 second intervals for the first five minutes and at 1 minute intervals for the next two hours. From Figure 6, we observe that the drift remains nearly constant after a short period of time for our experimental setup and for the idle Ubuntu and Vista systems, the drift stabilizes within a few minutes. The drift for a system running Mozilla Firefox was found to be nearly constant within 10 minutes. Running *tar* and *gzip* for compressing a large folder or *dd* to dump the contents of memory into a file resulted in most of the memory being changed within a minute due to second-order taint. To summarize, our tests demonstrated that there is no taint introduced in the extended memory by using Forenscope and that Forenscope can be used for forensic analysis where taint needs to be minimized.

Effectiveness against anti-forensics tools: Although forensics techniques can collect significant amounts of information, investigators must be careful to ensure the veracity and fidelity of the evidence collected because anti-forensic techniques can hide or intentionally obfuscate information gathered. In particular, rootkits can be used by hackers to hide the presence of malicious software such as bots running in the system. Malware tools such as the FU rootkit [16] directly manipulate kernel objects and corrupt process lists in ways that many tools cannot detect.

Table 5: Sizes of Forenscope and modules

Component	Lines of Code	Compiled Size (bytes)
Forenscope (C)	1690	15,420
Forenscope (Assembly)	171	327
Forenscope (Hardware)	280	1,441
Neutralizer & Forenshell	34	8,573
Other Modules	861	22,457
Total	3,036	48,218

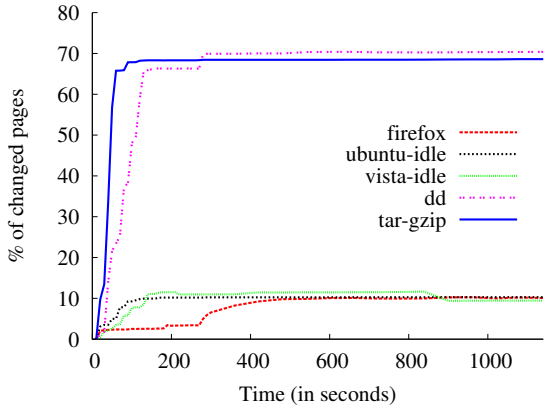


Figure 6: Comparison of Memory Blurriness

Malware researchers have also demonstrated techniques to evade traditional memory analysis through the use of low-level rootkits [28] which cloak themselves by deceiving OS-based memory acquisition channels on Linux and Windows. Hardware [12] and software [20] virtualization-based rootkits may be tricky to detect or remove by the legitimate operating system or application software because they operate one layer below standard anti-malware facilities. We describe and evaluate how Forenscope reacts to several publicly available rootkits. The set of rootkits was chosen to cover a gamut of representative threats, but the list is not meant to be exhaustive due to space constraints.

DR: The DR rootkit uses processor-level hardware debug facilities to intercept system calls rather than modifying the actual system call table itself. DR reprograms a hardware breakpoint which is reached every time a system call is made [15]. The breakpoint then intercepts the call and runs its own handler before passing control to the legitimate system call handler. Since Forenscope does not restore the state of debug registers, DR is effectively neutralized across the reboot, and as a result, hidden processes are revealed. Informant detects DR in several ways: DR is present in the module list, DR symbols are exported to the kernel and DR debug strings are present in memory. If an attacker modifies DR to make it more stealthy by removing these indicators, we contend that it is still hard to deceive Forenscope, since the debug registers are cleared as part of the reboot process. Although Forenscope doesn't restore the contents of the debug registers faithfully, this doesn't pose a problem for most normal applications because only debuggers typically use this functionality.

Phalanx B6: Phalanx hijacks the system call table by directly writing to memory via the */dev/mem* memory device. It works by scanning the internal symbol table of the kernel and redirecting control flow to its own internal functions. Informant detects Phalanx while checking the system call table and common kernel pointers. Neutralizer restores the correct pointers to inoculate Phalanx.

Adore: Adore [8] is a classic rootkit which hijacks kernel pointers to deceive tools such as *ps* and *netstat*. It works by overwriting pointers in the */proc* filesystem to redirect control flow to its own functions rather than modifying the syscall table directly. Informant detects that the pointers used by Adore do not belong to the original read-only program code segment of the kernel and Neutralizer restores the correct pointers. Restoration of the original pointers is simple and safe because the overwritten VFS function operations tables point to static functions such as *proc_readdir*, while Adore has custom handlers located in untrusted writable kernel module address space.

Mood-NT: Mood-NT is a versatile multi-mode rootkit that can hook the system call table, use debug registers and modify kernel pointers. Because of its versatility, the attacker can customize it for different purposes. Like the rootkits described previously, Forenscope detects Mood-NT in various modes. Our experiments indicate that Mood-NT hooks 44 system calls and Forenscope detects all of these alterations. Furthermore, each hook points out of the kernel's read-only program code address space and into the untrusted memory area occupied by the rootkit.

Size: Forenscope is written in a mixture of C and x86 assembly code. Table 5 shows that Forenscope is a very small program. It consumes less than 48 KB in code and 125 KB in running memory footprint. The lines of code reported in the table are from the output of the *sloccount* [29] program. We break down the size of each component into core C and assembly code, hardware-specific restoration code and module code. To minimize its size, Forenscope reuses existing kernel code to reinitialize the disk and network; the size of this kernel code is device-specific and therefore excluded from the table, since these components are not part of Forenscope. The small compiled size of Forenscope and its modules implies that a minimal amount of host memory is overwritten when Forenscope is loaded onto the system. Furthermore, the diminutive size of the code base makes it more suitable for auditing and verification.

5. DISCUSSION

While evaluating Forenscope, we observed different behavior of rootkits on virtual machines and physical hardware. Our observations confirm the results of Garfinkel et al [17] that virtual machines cannot emulate intricate hardware nuances faithfully and as a result some malware fails to activate on a virtual machine. For example, malware such as the Storm worm and Conficker [30] intentionally avoid activation when they sense the presence of virtualization to thwart the analysis process. Hence analyzing a system for rootkits using a virtual machine may not only cause some rootkits to slip under the radar but also alert them to detection attempts. Since Forenscope continues to run the system without exposing any of the issues raised by running virtualization systems, we argue that the system is unlikely to tip off an attacker to the presence of forensic software. Legally, the jury is still out on the use of live forensic tools because of the issues of taint and blurriness. While some recent cases [2] suggest that courts are starting to recognize the value of the contents of volatile memory, the validity of the evidence is still being contested. A recent manual on collecting evidence in criminal investigations released by

[8] http://stealth.openwall.net

Table 6: Effectiveness against rootkit threats

Rootkit	Description	Sanitization action
DR	Uses debug registers to hook system calls	Rebooting clears debug registers
Phalanx b6	Uses /dev/kmem to hook syscalls	Restore clean syscall table
Mood-NT	Multi-module RK using /dev/kmem/	Clear debug regs, restore pointers
Adore	Kernel module hooks /proc VFS layer	Restore original VFS pointers

the Department of Justice [6], instructs that *no limitations should be placed on the forensic techniques that may be used to search* and also states that use of forensic software, no matter how "sophisticated," does not affect constitutional requirements. Although we do not make strict claims of legal validity in the courts, we are encouraged by the above guidelines to collect as much volatile information as possible. We objectively compare our tool against the state of the art and find that it does collect more forms of evidence with better fidelity than existing tools.

Countermeasures: Although Forenscope provides deep forensic analysis of a system in a wide variety of scenarios, there are countermeasures that attackers and criminals can use to counter the use of Forenscope. From an incident response perspective, we assume that the machine is controlled by the owner and that the attacker does not have physical access to it. This means that only software-based anti-forensic techniques are feasible, although some of these techniques may involve changing hardware settings through software. Most of the hardware and software state involved in these anti-forensic techniques are cleared upon reboot or rendered harmless in Forenscope's clean environment. In investigation, the adversary may elect to use a BIOS password, employ a secure bootloader, disable booting from external devices or change BIOS settings to clear memory at boot time. These mitigation techniques may work, but if the investigator is sophisticated enough, he can try techniques suggested by Halderman et al [19] to cool the memory chips and relocate them to another machine which is configured to preserve the contents of DRAM at boot time. One other avenue for working around a password-protected BIOS is to engage the bootloader itself. We found that some bootloaders such as GRUB allow booting to external devices even if the functionality is disabled in the BIOS. The only mitigation against this channel is use password protection on GRUB itself, which we believe is not frequently used.

Limitations: The only safe harbor for malware to evade Forenscope is in conventional memory itself because the act of rebooting pollutes the contents of the lower 640 KB of memory considerably thus potentially erasing evidence. However, we contend that although this technique is possible, it is highly unlikely for three reasons: first, for such malware to persist and alter the control flow, the kernel must map in this memory area in the virtual address space. This requires a change in the system page tables which is easily detectable by Forenscope since most modern operating systems do not map the conventional memory space into their virtual memory space. Secondly, such malware would have to inject a payload into conventional memory and if the payload is corrupted by the reboot process, the system will crash. Finally, such malware won't survive computer hibernation because conventional memory is not saved in the process. Even if Forenscope is unable to restore the system due to extenuating circumstances, we still have an intact memory dump and disk image to analyze. Although Forenscope has been designed with investigation in mind, we have not designed it to be completely transparent. For instance, malware might detect the presence of Forenscope by checking BitBlocker write latencies or scanning conventional memory.

6. RELATED WORK

Forenscope uses many technologies to achieve a high fidelity forensic analysis environment through introspection, data structure analysis and integrity checking. Many of the introspective techniques used by Forenscope were inspired by similar functionality in debuggers and simulators. VMware's VMsafe protects guest virtual machines from malware by using introspection. A virtual machine infrastructure running VMsafe has a security monitor which periodically checks key structures in the guest operating system for alteration or corruption. Projects such as Xenaccess [22] take the idea further and provide a way to list running processes, open files and other items of interest from a running virtual machine in a Xen environment. Although Xenaccess and Forenscope provide similar features, Xenaccess depends on the Xen VMM, but the investigator cannot rely on its presence or integrity. On some older critical infrastructure machines, legacy software requirements make it impractical to change the software configuration. Forenscope does not have such requirements. Forenscope's techniques to recover operating system state from structures such as the process list have been explored in the context of analyzing memory dumps using data structure organization derived from reverse-engineered sources [14, 27]. Attestation shows that a machine is running with an approved software and hardware configuration by performing an integrity check. Forenscope builds upon work from the VM introspection community to allow forensic analysis of machines that are not prepared a priori for such introspection. It provides a transparent analysis platform that does not alter the host environment and Forenscope supports services such as BitBlocker that allow an investigator to explore a machine without inducing taint.

The techniques used by Forenscope for recovering running systems are well grounded in the systems community and have been studied previously in different scenarios. The original Intel 286 design allowed entry into `protected mode` from `real mode`, but omitted a mechanism to switch back. Microsoft and IBM used an elegant hack involving memory remanence to force re-entry into real mode by causing a reboot to service BIOS calls. This technique was described by Bill Gates as "turning the car off and on again at 60 mph" [24]. Some telecommunications operating systems such as Chorus [25] are designed for quick recovery after a watchdog reset and simply recover existing data from the running operating system rather than starting afresh. David [13] showed that it is possible to recover from resets triggered by the watchdog timer on cell phones. BootJacker [11] showed that it is possible for attackers to recover and compromise a running operating system by using a carefully crafted forced reboot. Forenscope applies these techniques in the context of forensic analysis and our work presents the merits and limitations of using such techniques to build a forensic tool.

Devices such as the Trusted Platform Module and Intel trusted execution technology (TXT) provide boot time and run-time attestation respectively. Although TPM may be available for some machines, the protection afforded by a TPM may not be adequate for machines which are meant to run continuously for months. These machines perform an integrity check when they boot up, but their lengthy uptime results in a long time of check to time of use (TOCTTOU) that extends the duration for breaches to remain undetected. Hardware solutions such as Copilot [23] are available to check system integrity. In contrast, Forenscope performs an integrity assessment at the time of use; which allows the investigator to collect evidence with better fidelity.

7. CONCLUDING REMARKS

Forenscope explores live forensic techniques and the issues of evidence preservation, non-intrusiveness and fidelity that concern such approaches. Measured against existing tools, our experiments show that Forenscope can achieve better compliance within the guidelines prescribed by the community. Forenscope shows that volatile state can be preserved and the techniques embodied in Forenscope are broadly applicable. We encourage further development of tools based on our high-fidelity analysis framework and believe that it can enable the advancement of analysis tools such as KOP [9]. Extensive evaluation of our techniques has shown that they are safe, practical and effective by minimally tainting the system, while causing no disruption to critical systems. We believe that these techniques can be used in cases where traditional tools are unable to meet the needs of modern investigations. To continue the development of this tool, we plan to work closely with partners to better evaluate use of this tool in real-world scenarios such as incident response in a variety of contexts.

Acknowledgements We would like to thank the anonymous reviewers, Winston Wan, Mirko Montanari and Kevin Larson for their valuable feedback. This research was supported by grants from DOE DE-OE0000097 under TCIPG (tcip.iti.illinois.edu) and a Siebel Fellowship. The opinions expressed in this paper are those of the authors alone.

8. REFERENCES

[1] SANS Top 7 New IR/Forensic Trends In 2008. `http://computer-forensics.sans.org/community/top7_forensic_trends.php`.

[2] Columbia Pictures Indus. v. Bunnell, U.S. Dist. LEXIS 46364. C.D. Cal. `http://www.eff.org/cases/columbia-pictures-industries-v-bunnell`, 2007.

[3] *Prosecuting Computer Crimes*, pages 141–142. US Department of Justice, 2007.

[4] Electronic Crime Scene Investigation: A Guide for First Responders. pages 25–27, 2008.

[5] Ramdisks - Now We are Talking Hyperspace! `http://www.linux-mag.com/cache/7388/1.html`, 2009.

[6] *Searching and Seizing Computers and Obtaining Electronic Evidence in Criminal Investigations*, pages 79,89. Computer Crime and Intellectual Property Section Criminal Division, 2009.

[7] K. Amari. Techniques and Tools for Recovering and Analyzing Data from Volatile Memory, 2009.

[8] D. Brezinski and T. Killalea. Guidelines for Evidence Collection and Archiving. RFC 3227 (Best Current Practice), Feb. 2002.

[9] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 555–565, New York, NY, USA, 2009. ACM.

[10] C. C. Center. How the FBI Investigates Computer Crime. `http://www.cert.org/tech_tips/FBI_investigates_crime.html`, 2004.

[11] E. Chan, J. Carlyle, F. David, R. Farivar, and R. Campbell. BootJacker: Compromising Computers using Forced Restarts. In *Proceedings of the 15th ACM conference on Computer and Communications Security*, pages 555–564. ACM New York, NY, USA, 2008.

[12] D. Dai Zovi. Hardware Virtualization Rootkits. *BlackHat Briefings USA, August*, 2006.

[13] F. M. David, J. C. Carlyle, and R. H. Campbell. Exploring Recovery from Operating System Lockups. In *USENIX Annual Technical Conference*, Santa Clara, CA, June 2007.

[14] B. Dolan-Gavitt. The VAD tree: A Process-eye View of Physical Memory. *Digital Investigation*, 4:62–64, 2007.

[15] Edge, Jake. DR rootkit released under the GPL. `http://lwn.net/Articles/297775/`.

[16] Fuzen Op. The FU rootkit. `http://www.rootkit.com/project.php?id=12`.

[17] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is not transparency: VMM detection myths and realities. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS-XI)*, May 2007.

[18] P. Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *Proceedings of the 6th USENIX Security Symposium*, pages 77–90, July 1996.

[19] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, and J. A. Calandrino. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *Proc of the 17th USENIX Security Symposium*, San Jose, CA, July 2008.

[20] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing malware with virtual machines. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 314–327, Washington, DC, USA, 2006. IEEE Computer Society.

[21] W. Link and H. May. Eigenshaften von MOS-Ein-Transistorspeicherzellen bei tieften Temperaturen. In *Archiv fur Elektronik und Ubertragungstechnik*, pages 33–229–235, June 1979.

[22] B. Payne, M. de Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of 23rd Annual Computer Security Applications Conference*, pages 385–397, 2007.

[23] N. Petroni, T. Fraser, J. Molina, and W. Arbaugh. Copilot-A Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium*, pages 179–194, 2004.

[24] J. Pournelle. OS | 2: What is is, What is isn't – and some of the Alternatives. *Infoworld*, 1988.

[25] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Lonard, and W. Neuhauser. Overview of the CHORUS Distributed Operating Systems. *Computing Systems*, 1:39–69, 1991.

[26] A. Savoldi and P. Gubian. Blurriness in Live Forensics: An Introduction. In *Proceedings of Advances in Information Security and Its Application: Third International Conference, Seoul, Korea*, page 119. Springer, 2009.

[27] A. Schuster. Searching for Processes and Threads in Microsoft Windows Memory Dumps. The Proceedings of the 6th Annual Digital Forensics Research Workshop, 2006.

[28] S. Sparks and J. Butler. Raising The Bar for Windows Rootkit Detection. *Phrack*, 11(63), 2005.

[29] D. A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount`.

[30] B. Zdrnja. More tricks from Conficker and VM detection. `http://isc.sans.org/diary.html?storyid=5842`, 2009.

A Multi-User Steganographic File System on Untrusted Shared Storage

Jin Han Meng Pan Debin Gao HweeHwa Pang

Singapore Management University

{jin.han.2007, mengpan, dbgao, hhpang}@smu.edu.sg

ABSTRACT

Existing steganographic file systems enable a user to hide the existence of his secret data by claiming that they are (static) dummy data created during disk initialization. Such a claim is plausible if the adversary only sees the disk content at the point of attack. In a multi-user computing environment that employs untrusted shared storage, however, the adversary could have taken multiple snapshots of the disk content over time. Since the dummy data are static, the differences across snapshots thus disclose the locations of user data, and could even reveal the user passwords.

In this paper, we introduce a Dummy-Relocatable Steganographic (DRSteg) file system to provide deniability in multi-user environments where the adversary may have multiple snapshots of the disk content. With its novel techniques for sharing and relocating dummy data during runtime, DRSteg allows a data owner to surrender only some data and attribute the unexplained changes across snapshots to the dummy operations. The level of deniability offered by DRSteg is configurable by the users, to balance against the resulting performance overhead. Additionally, DRSteg guarantees the integrity of the protected data, except where users voluntarily overwrite data under duress.

1. INTRODUCTION

Steganographic File Systems (stegfs) are intended to provide plausible deniability to data owners in the event that they are forced to disclose their secret data [4]. A stegfs hides encrypted user data among dummy data that contain only pseudo-random bits. Without the correct password, it is not possible to differentiate user data from dummy (based on the assumption that the output of the block cipher is indistinguishable from random bits [3, 4]), even for an adversary who understands the mechanisms of the file system and is able to gain access to the storage devices. This feature allows a data owner to selectively reveal some directories/files, but disclaim the existence of his sensitive data.

To be believable, the disclaimer of the data owner must be consistent with the information that the adversary is able to gather about the file system. This is much more challenging to achieve in modern computing environments when the user data are encrypted and stored in shared network storage. Compared to portable and local storage, network storage dramatically increases the availability and accessibility of user data. However, it also brings new challenges in securing user data. With shared network storage, the adversary is no longer limited to a single snapshot of the disk content at the point of attack. Instead, the adversary could now locate the physical server machines being used [17] and quietly amass multiple snapshots of the file system over a period of time before launching his attack. The additional knowledge that the adversary gleams from the multiple snapshots must be factored into the stegfs design.

In earlier stegfs designs [4, 15, 12, 16], dummy data are created when the disk is formatted and remain static thereafter. These schemes are effective against adversaries who only see the final state of the storage, but cannot defend against adversaries who possess multiple snapshots of the storage. Indeed, changes among different snapshots not only reveal the location of secret data, but could even be utilized to recover the access keys (for example, when the first scheme by Anderson et al. [4] is utilized). Recent stegfs schemes, which are proposed to defend against multiple-snapshots attacks, either cannot guarantee the integrity of user data even under legitimate data operations [8, 9], or require a trusted agent to manage all the user passwords and dummy data [20], which effectively presents a single point of disclosure for user passwords.

In this paper, we propose a multi-user stegfs for shared storage systems, which is named as DRSteg – Dummy Relocatable Steganographic file system. DRSteg is designed to meet the following requirements:

- Security: To provide plausible deniability of secret data in a multi-user environment in which the adversary could obtain multiple snapshots of the storage content. This protection should extend to any user even when the storage server and all the other users are completely compromised, i.e., they have surrendered all the information in their possession.

- Usability: To guarantee data integrity, and at the same time enable individual users to trade off between deniability and system performance.

To the best of our knowledge, DRSteg is the first stegfs that allows I/O operations observed on shared storage to

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

be plausibly attributed to dummy data without requiring a trusted agent as used by Zhou et al. [20]. In addition, our work also manages to increase the deniability provided to individual users by sharing dummies among multiple users in the system. It is technically challenging to satisfy both the security and usability requirements, especially when dummies are shared. DRSteg incorporates a special dummy relocation mechanism that enables individual users to distinguish dummies from other users' data (in order to free dummies without destroying data), and to prevent adversaries from discerning the difference between dummy and user data even after obtaining multiple snapshots.

This is also the first work that formalizes the deniability achieved by a multi-user stegfs. The formalization enables us to develop a tunable mechanism for users to balance between deniability and system responsiveness. In DRSteg, the deniability enjoyed by individual users could be maintained beyond a specified threshold, whether or not all the other users are fully compromised. The amount of dummy operations is controlled individually; a user who specifies a more aggressive amount enjoys higher deniability at the expense of slower file operations.

To substantiate the usability of DRSteg, we present results of an empirical evaluation using file operation logs collected from 12 graduate students in our school. The results confirm that DRSteg is capable of achieving a wide range of user-specified deniability levels. We also implemented a prototype of DRSteg as a file system module in Linux kernel. Performance experiments on the prototype show that security and performance can be traded off against each other.

2. RELATED WORK

Cryptographic file systems (e.g., [5, 7, 11, 19]) and their implementations (e.g., [1, 2]) have been studied extensively in the last two decades . A cryptographic file system complements the access control mechanism of the operating system (OS). Even if the OS is compromised or the data storage is removed from the OS, data in the file system remain protected by the user's password. A weakness of cryptographic file systems is that they leave evidence of the existence of encrypted data, so a determined attacker may compel the users to reveal their decryption passwords.

In order to provide plausible deniability of the existence of secret data, Anderson et al. proposed two steganographic file system (stegfs) schemes [4]. In the first scheme, the disk is initialized with several cover files that have equal length and contain random data. A secret object is stored through an exclusive-or operation on a subset of the cover files, identified by the corresponding bits in the access key. To protect against brute force attacks, the number of cover files must be sufficiently large; this imposes heavy I/O overheads as each read/write request for an object translates into operations on multiple cover files. The scheme is effective against single-snapshot attacks but not multiple-snapshot attacks. In particular, the differences between just two snapshots of the storage can expose the access key used[1].

In Anderson's second scheme [4], the disk is first filled with random bits. Subsequently, secret data blocks are written to pseudorandom addresses. An implementation of this scheme on Linux is reported by McDonald et al. [15], a peer-to-peer version by Hand et al. [12] and a distributed version by Giefer et al. [10]. The disadvantage of the scheme is that the probability of collision in the locations where data are stored increases as more data are added to the disk. Although replicating each data block in different locations reduces the likelihood of data loss, the risk cannot be eliminated; hence data integrity is not guaranteed.

Pang et al. [16] utilized a bitmap to track block allocation to avoid overwriting data and to improve system performance. To defend against single-snapshot attacks, dummy data are added when the disk is initialized. The dummy data cannot be changed or relocated at runtime, so the scheme is susceptible to multiple-snapshot attacks. Zhou et al. [20] provided for the relocation of dummy blocks. Their solution requires a trusted agent to manage all the user passwords and dummy data, which effectively transfers the risk of password disclosure to the agent.

Diaz et al. [8] proposed to defend against traffic analysis [18] through a mix-based stegfs that employs a local mix to relocate files in the remote storage. They show that the security of the scheme depends on the file-size patterns in the system. Another work by Domingo-Ferrer et al. [9] addressed the problem of data loss in a stegfs with multiple users. It is not designed to defend against multiple-snapshot attacks though. Furthermore, neither of the two schemes guarantees data integrity under legitimate data operations.

TrueCrypt[2], an open-source disk-encryption software package, enables a user to create a deniable file system within a regular encrypted file or partition. The file system is deniable if the adversary only sees the final content of the disk. However, it cannot defend against an adversary who possesses multiple snapshots of the encrypted partition. The same weakness exists in similar products that provide deniability for secret files, e.g., Phonebook[3] and Rubberhose[4].

Note that deniability in stegfs is different from deniable encryption [6] which allows an encrypted message to be decrypted into different sensible plaintexts with different keys. Stegfs is also different from private information retrieval (PIR) [13] which allows a user to retrieve an item from a server without revealing which item is retrieved. A stegfs allows the untrusted server to be cognizant of which disk blocks are retrieved, yet provides deniability that they stemmed from operations on secret data. A stegfs is not designed to prove non-existence of secret data but to provide plausible deniability of the existence of secret data.

3. PROBLEM DEFINITION

3.1 Threat Model

Figure 1 depicts our model of a multi-user file system. In the model, user data are stored on a shared storage. The stegfs functionalities are implemented in the client module that runs on the user computers. This client module is secured so that sensitive data that are operated on as well as any passwords used for encrypting and decrypting the data are protected. The storage server manages the shared stor-

[1]This is because only cover files whose indexes correspond to bits with value "1" in the access key will be modified for any data modification. Those files which do not change will correspond to bits with value "0" in the access key. Thus, the access key can be reconstructed by observing the changes of the cover-file matrix in the storage.

[2]TrueCrypt, `http://www.truecrypt.org/`

[3]Phonebook, `http://www.freenet.org.nz/phonebook`

[4]Rubberhose, `http://iq.org/~proff/rubberhose.org`

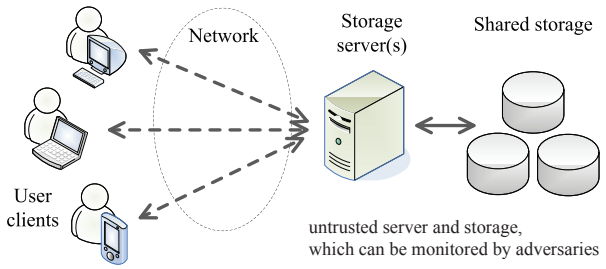


Figure 1: A multi-user stegfs with untrusted shared storage

age devices which provide block-level operations, including DAS (direct attached storage) and SAN (storage area network). Different from the model where the server manages all the user passwords [20], the storage server and shared storage in our model are not stegfs specific.

The server and the storage devices are not trusted. This means that an adversary may infiltrate the server or the storage devices directly (or the backup of these devices) to copy and analyze the stored content. Although our scheme provides better protection when the communication between users and the server is anonymized, it is not a necessary condition for DRSteg to provide deniability to users. We will analyze the deniability of DRSteg under different scenarios in Section 5.

In this paper, we focus on adversaries who are after the user data, and we explicitly rule out considerations of sabotage like overwriting/deleting data and denial of service. The threat posed by the adversary thus hinges on two factors: (a) his knowledge of the file system state, and (b) his access to the users of the system. These two factors together determine the adversary's ability to make deductions about the hidden data on the storage, and to verify any claims elicited from the users.

The first factor, knowledge of the storage state, is characterized by the number of observations of the storage content. An adversary who is able to access the storage only once (i.e., at the point of attack) only gains a *single snapshot* of the storage. An example is someone who is captured by criminals and forced to reveal all the contents in his portable drive. However, when the adversary has more than one chance to access the storage, he can record *multiple snapshots*. The information in those snapshots is then utilized to deduce the existence of secret data.

The second factor that defines the adversary's ability concerns his access to the users. Here, we make the following assumption:

Victim isolation assumption. In coercing information from the users, it would be effective for the adversary to interrogate them separately and cross-check the information elicited. Placed in isolation, a victim knows neither which other users have been compromised nor what information they have surrendered. Consequently, each victim has to assume the worst, i.e., that all the other users are compromised and all their secrets are revealed. He thus has to independently decide what data he can hide without being contradicted by other users' disclosure.

Multi-user encrypting file systems [2, 5, 7] are inadequate under the victim isolation assumption, as it is not safe for a user to claim his data to belong to someone else. A solution is to use dummy blocks, which should be operated on in similar ways as encrypted data blocks in order to defend against multiple snapshot attacks.

3.2 Definition of Deniability

To formalize the threat, an adversary has access to a sequence of snapshots $\mathsf{S} = \{\mathsf{s}_1, \mathsf{s}_2, \ldots, \mathsf{s}_T\}$ of the stegfs partition on the disk, where s_T is the snapshot at the time of coercion. Following the victim isolation assumption, the adversary extracts all the passwords from other users (P') at the time of attack, and also coerces the victim to reveal his passwords $\mathsf{P}_t = \{\mathsf{p}_1, \mathsf{p}_2, \ldots, \mathsf{p}_t\}$. The adversary then utilizes the passwords obtained to decode the information in each snapshot.

Let $\mathsf{H}_i^{\mathsf{dummy}}$ and $\mathsf{H}_i^{\mathsf{data}}$ denote the hypotheses that an allocated block blk_i is a dummy block and a data block, respectively. Let e_i denote the evidence on blk_i observed from S, and $\mathsf{E} = \{\mathsf{e}_i\}$ the aggregate evidence across all the disk blocks. We define the *plausible deniability* of blk_i as follows.

Definition 1. *Given the evidence* $\mathsf{E} = \{\mathsf{e}_i\} = \mathsf{S} \cup \mathsf{P}' \cup \mathsf{P}_t$, *where* $\mathsf{S} = \{\mathsf{s}_1, \mathsf{s}_2, \ldots, \mathsf{s}_T\}$ *is a sequence of snapshots taken by the adversary and* $\mathsf{P}' \cup \mathsf{P}_t$ *is the set of passwords revealed to the adversary (along with the blocks decrypted with these passwords), the deniability of an allocated block* blk_i *is the posterior probability that* e_i *was generated by operations on dummy block* blk_i*:*

$$\mathsf{deny}_i = Pr(\mathsf{H}_i^{\mathsf{dummy}} | \mathsf{e}_i) \quad (1)$$

A steganographic file system is said to be ***α-deniable*** *if*

$$\mathsf{deny}_i \geq \alpha$$

for all blk_i *that cannot be decrypted with* $\mathsf{P}' \cup \mathsf{P}_t$*, for any* $t \geq 1$ *of the user's choice.*

An α-deniable stegfs guarantees that any evidence gathered by an adversary (e.g., disk images across multiple snapshots) is caused by dummy data operations with at least a probability of α. This means that a user of the system can attribute the evidence to dummy operations without revealing his secret data.

4. DESIGN OF DRSteg

DRSteg is designed to enable a user to selectively disclose some of his data, while enjoying α-deniability for the rest of the data that he is withholding from the adversary. We begin this section with an overview of the DRSteg design, before presenting the detailed data structures and implementation considerations.

4.1 Overview of DRSteg

In DRSteg, each user must be able to protect his data with different passwords, so that he can surrender some data but not others. To achieve α-deniability for the data blocks that he is withholding, our approach is to (a) enforce a joint ownership for allocated disk blocks to prevent the adversary from associating with certainty a withheld block with any particular user, and (b) introduce dummy blocks that are operated on at runtime, so that changes to the withheld blocks can be plausibly explained by dummy operations.

We realize the joint ownership through a voting protocol. For every allocated block, m ownership shares are created and distributed to m users, including the user who requested

for the block (also known as the creator). A block can subsequently be altered or freed only after all the m shares have been garnered from consenting owners. By following this policy, we ensure that the block is never deallocated without the creator's share, yet the creator of the block is obfuscated among the share owners. The creator may use an allocated block either for his data or as a dummy.

For each user, the disk blocks that hold his data are protected by one of his passwords $p_1, p_2, \ldots, p_n$. The number of passwords n is expected to vary from user to user, though we use the same symbol n across users for brevity. Moreover, the passwords are generated as a hash chain [14], i.e., $p_l = h(p_{l+1})$ for a hash function h and $1 \leq l < n$ (as illustrated in the upper part of Figure 2). By supplying any password p_l, $1 \leq l \leq n$, the user can access all the secret data at and below level l.

As for those disk blocks that are allocated as dummies, no bookkeeping information is maintained to track them directly; otherwise, the adversary can simply demand the bookkeeping information from the users, and with it discover the dummy blocks in the file system. Instead, a dummy block can only be identified through the cooperation of its owners: Each shareholder of the block checks whether it is protected with one of his passwords; if not, the block is a potential dummy – it may indeed be a dummy, or it may hold the data of some other user. It is freed in the same way as data blocks, i.e., after gathering m shares.

In the event of an attack, our DRSteg design allows a coerced user to supply some password p_t, $1 \leq t < n$, to the adversary and deny the existence of the passwords p_j for $t < j \leq n$. The data blocks that are protected by p_j then appear to be potential dummies, thus enabling the user to hide the existence of the data.

4.2 Detailed Design of DRSteg

Drawing on the approaches introduced above, we now put together the concrete DRSteg design. Each user u keeps track of a set of blocks A_u on which he currently holds a share. Moreover, each password p_l protects a set of data blocks $D_{u,l}$. The set difference $A_u - \cup_l D_{u,l}$ gives the blocks that exclude u's data, and dummy blocks are the allocated blocks that contain nobody's data, i.e., $\cap_u(A_u - \cup_l D_{u,l})$. Figure 2 depicts our detailed design for DRSteg (the encryption is done at the granularity of individual blocks).

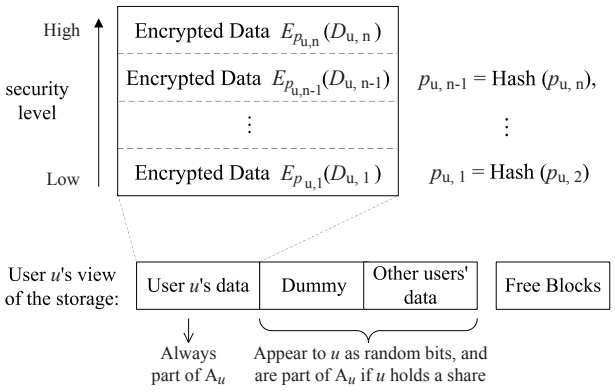


Figure 2: Key management and user view of the storage

Whenever a user u requires a disk block blk from the file system to write data or dummy patterns, a free disk block is allocated and shares of the block are also created. One share is given to u, while the remaining shares of blk are distributed to other users u', i.e., $A_u \leftarrow A_u \cup \{\mathsf{blk}\}$ and $A_{u'} \leftarrow A_{u'} \cup \{\mathsf{blk}\}$. If user u encrypts data with his password p_l and stores it in blk, then $D_{u,l} \leftarrow D_{u,l} \cup \{\mathsf{blk}\}$.

Any user u may propose the deletion of a block in his A_u. The deletion is effected only after all the users who hold shares of the block have acquiesced. Obviously, if the block holds the data of user u', he would relocate the data before supporting the deletion. This is to avoid leaving clues for differentiating between dummy and data blocks.

With DRSteg, user u can surrender any password p_t, $1 \leq t < n$ and claim that data blocks in $A_u - \cup_{1 \leq j \leq t} D_{u,j}$ are not his data. Claiming that data blocks in $D_{u,j}$ for $t < j \leq n$ are dummy blocks is plausible since they also appear in $A_{u'} - \cup_l D_{u',l}$ of other users u' who hold shares of the blocks.

4.2.1 Joint ownership of blocks

We implement the joint ownership of disk blocks through a voting protocol and two data structures – a set of encrypted user share boxes (USB) and a global voting table (GVT) in clear text. A USB is used to track the A_u of each user, and a GVT records the votes surrendered by users. Two other structures are additionally maintained in clear text in the storage: a list of the users' public keys, and a bitmap to track the allocation status of the disk blocks.

When a user allocates a disk block blk_i, he 1) sets the bit of this block to "1" in the bitmap; 2) creates m shares and writes them to the corresponding USBs; 3) writes the encrypted/random data content to the block. The format of each encrypted share is given as $E(K_{pub,u}, i)$, an encryption of i with a user's public key. The encrypted shares denote the ownership of this block. A block $\mathsf{blk} \in A_u$ if the share $E(K_{pub,u}, i)$ exists in the USB of user u. The m owners of a block include the creator and $m-1$ other users randomly selected from the public-key list.

Any of the m owners can subsequently initiate the deletion of the block blk by writing i to the global voting table (GVT) and removing his share from his USB. To support the deletion, other owners also contribute their shares into GVT. When the number of accumulated shares of a block reaches m, this block can be removed from GVT and its bit in the bitmap is set to "0" (indicating that this block is free). The share constitution ensures that the block can be deallocated only when block creator signals his agreement by surrendering his share to the GVT.

4.2.2 Management of data blocks

In order to provide plausible deniability against multiple-snapshot attacks, disk blocks that contain data must be managed carefully so that they leave the same evidence as operations on dummy blocks.

First, consider the modification of secret data. By comparing snapshots, the adversary may discover that the content of a block changes before all the m shares are added into GVT. This would never happen to a dummy block according to our voting protocol. Therefore, instead of overwriting data blocks, each user always migrates his updated content to new blocks, and initiates the deletion of the outdated blocks in GVT so that they will be freed in due course. However, the initiation of the deletion operation is delayed,

in order to break the temporal correlation between the allocation of new blocks and the deallocation of outdated blocks.

Next, consider the case where some user's data block is registered for deallocation in GVT by other users. If the user never concurs, the adversary will suspect that the block contains data, since deallocation of dummy blocks are supported readily. To avoid suspicion, the user has to migrate the content to a fresh disk block, before relinquishing his share to the old data block.

In real implementations, the block creating operations are carried out immediately, but the voting (including removing shares from USB and writing block numbers into GVT) are delayed. We pass the voting operations to a background user process that survives beyond user log-off. The background process repeatedly initiates the deletion of a block in its pool after sleeping for a random duration. This makes the operations for data blocks plausible since the creation and voting could be caused by either creating and freeing dummy blocks or creating, modifying and freeing data blocks.

4.3 Discussions

4.3.1 Comparing to naive designs

There also exist alternatives in designing a multi-user steganographic file system. A naive one could simply let each user manage his own blocks (including data and dummy). Since dummy blocks are no longer shared, one has to create many more dummy blocks in order to achieve the same deniability compared to our design, when anonymous channels are used between the users and the storage server. When this channel is not anonymized, our design still provides similar security and disk utilization compared to the naive design. The deniability provided by DRSteg under both scenarios is analyzed in the next section.

4.3.2 Encryption of the block shares

Another security issue relates to the encryption of the shares in USB. If the shares are stored in clear text, it will be straightforward for an adversary to identify who the owners of any particular block are. By encrypting the shares, the owners of any block are obfuscated so long as multiple blocks have been allocated between snapshots. In this way, our approach safeguards shareholders from being earmarked to be the next target of coercion.

4.3.3 Organization of the user passwords

The last design issue concerns the organization of the user passwords. One option is to have only one password in each account and to give every user multiple accounts. Under coercion, a user reveals some of his accounts and tries to hide the remaining ones. However, this simple option fails when the adversary captures all the users of the system. When that happens, the adversary can check whether there are m shares among the surrendered accounts for every allocated block; if not, there must exist more user accounts. This is why we choose to allow multiple passwords (for different security levels) in each user account.

Organizing multiple passwords in a hash chain has been proposed in other stegfs [4, 10, 16], and its one-way property meets our requirements well. Under coercion attack, the disclosure from surrendering t independent passwords is the same as giving up the t lowest-level passwords in a hash chain. Thus, in our system design, the hash chain mechanism is chosen due to the performance and usability benefits gained compared to independent passwords.

5. PLAUSIBLE DENIABILITY OF DRSteg

Having introduced the design of DRSteg, we now quantify the deniability it provides under a spectrum of progressively challenging attack scenarios. Based on the last and most demanding scenario, we then show how to operationalize the DRSteg design so as to sustain the system security above user-specified deniability thresholds. Table 1 summarizes the terms and notations which are used in the analysis.

5.1 Analysis of Deniability

We first expand Equation 1.

$$\mathsf{deny_i} = \Pr(\mathsf{H_i^{dummy}}|\mathsf{e_i}) = \frac{\Pr(\mathsf{e_i}|\mathsf{H_i^{dummy}}) \times \Pr(\mathsf{H_i^{dummy}})}{\Pr(\mathsf{e_i})} \quad (2)$$

According to our problem formulation in Section 3, the adversary is capable of taking multiple snapshots of the storage content. He may also augment the snapshots with secrets that he coerced from one or more users. The following attack scenarios differ on the amount of secrets thus extracted, and deserve particular attention in deploying DRSteg. These scenarios will be further evaluated in Section 6. In the following analysis, we consider the case where the evidence contains two snapshots. The analysis extends easily to multiple snapshots. Note that Equation 2 implicitly takes the frequency of these snapshots into consideration by evaluating $\mathsf{e_i}$, i.e., the more frequently snapshots are taken, the more information $\mathsf{e_i}$ would include.

5.1.1 Passive-adversary scenario

In this scenario, the adversary may be curious and has not resorted to force, or he may not be ready to expose himself just yet. Thus he only relies on the snapshots collected, i.e., the evidence $\mathsf{E} = \mathsf{S}$. By comparing any two recorded snapshots ($\mathsf{s_1}$, $\mathsf{s_2}$), the adversary could observe a lot of user activities, e.g., new blocks being created, deleted, and etc.

Let us first consider the creation of new blocks. A block $\mathsf{blk_i}$ is created between $\mathsf{s_1}$ and $\mathsf{s_2}$ if $\mathsf{flag_i}$ changes from 0 in $\mathsf{s_1}$ to 1 in $\mathsf{s_2}$. Let $\mathsf{crt^{data}}$ represent the net number of data blocks created between $\mathsf{s_1}$ and $\mathsf{s_2}$, and $\mathsf{crt^{dummy}}$ the net number of dummy blocks created in the same period. $\mathsf{ttl_{s_2}}$, $\mathsf{ttl_{s_2}^{dummy}}$, and $\mathsf{ttl_{s_2}^{data}}$ denote, respectively, the total number of allocated blocks, the total number of dummy blocks, and the total number of data blocks in $\mathsf{s_2}$. Given an evidence that $\mathsf{blk_i}$ is newly allocated, the probability that $\mathsf{blk_i}$ is a dummy block in $\mathsf{s_2}$ is calculated with Equation (2) as

$$\mathsf{deny_i} = \frac{\mathsf{crt^{dummy}}}{\mathsf{ttl_{s_2}^{dummy}}} \times \frac{\mathsf{ttl_{s_2}^{dummy}}}{\mathsf{ttl_{s_2}}} \Big/ \frac{\mathsf{crt^{data}} + \mathsf{crt^{dummy}}}{\mathsf{ttl_{s_2}}} = \frac{\mathsf{crt^{dummy}}}{\mathsf{crt^{data}} + \mathsf{crt^{dummy}}}$$

This derivation extends to block deletion and other evidence listed in Table 2. Denoting the number of data/dummy block operations between $\mathsf{s_1}$ and $\mathsf{s_2}$ by $\mathsf{op^{data}}$ and $\mathsf{op^{dummy}}$, the deniability can be calculated as $\mathsf{op^{dummy}}/(\mathsf{op^{data}} + \mathsf{op^{dummy}})$.

For an individual user u in DRSteg, let $\mathsf{op_u^{data}}$ denote the number of data blocks operated on in $\cup_l D_{u,l}$ between $\mathsf{s_1}$ and $\mathsf{s_2}$, and $\mathsf{op^{dummy}}$ denote the number of dummy blocks operated on in the system. The deniability that DRSteg provides for u under this scenario is expressed as

$$\mathsf{deny_{u,i}} = \frac{\mathsf{op^{dummy}}}{\mathsf{op_u^{data}} + \mathsf{op^{dummy}}} \quad (3)$$

Notation	Explanation
$\mathsf{S} = \{\mathsf{s}_1, \mathsf{s}_2, \ldots, \mathsf{s}_T\}$	Snapshots (of the stegfs partitions) taken by the adversary.
$\mathsf{P}_t = \{\mathsf{p}_1, \mathsf{p}_2, \ldots, \mathsf{p}_t\}$	Passwords revealed to the adversary under coercion.
$\mathsf{E} = \{\mathsf{e}_i\} = \mathsf{S} \cup \mathsf{P}' \cup \mathsf{P}_t$	Evidence possessed by the adversary.
$\mathsf{s}_k = \{\mathsf{BLK}, \mathsf{USB}, \mathsf{GVT}\}_k$	$\mathsf{BLK} = \{\mathsf{blk}_i\}$: Blocks in the stegfs partition (blk_i is the i-th block). $\mathsf{USB} = \{\mathsf{USB}_u\}$: User share boxes ($\mathsf{USB}_u$ is the USB of user u). GVT: Global voting table.
$\mathsf{blk}_i = \langle \mathsf{text}_i, \mathsf{flag}_i \rangle$	text_i: If blk_i is dummy, text_i contains random bits; If blk_i holds user data, $\mathsf{text}_i = E(\mathsf{p}, \mathsf{plaintext}_i)$ flag_i: A flag indicating whether blk_i has been allocated.
$\mathsf{H}_i^{\mathsf{dummy}}$, $\mathsf{H}_i^{\mathsf{data}}$	Hypothesis that blk_i is a dummy/data block in s_T.

Table 1: Summary of notations used

Evidence	DRSteg operation
flag_i changes from 0 to 1 and new shares appear in some USBs	Create blk_i as a new dummy or data block
A share of blk_i is moved from USB_u to GVT	User u votes to delete blk_i
flag_i changes from 1 to 0, and blk_i's entry is removed from GVT	Delete blk_i as enough votes are present in GVT
Some combination of the above	Some combination of the above

Table 2: Evidences and the corresponding DRSteg operations

5.1.2 *Anonymous-channel scenario*

Once the adversary starts to coerce users, by the victim isolation assumption in Section 3, one has to assume that all of the users have been captured and be wary about offering conflicting information to the adversary. In this scenario, we consider a victim u who discloses the passwords for up to level t of his files and attempts to hide his remaining data, when all the other users are compromised ($\mathsf{E} = \mathsf{S} \cup \mathsf{P}' \cup \mathsf{P}_t$). We assume that all the user requests where sent through an anonymous channel to the storage server, so that the adversary is not able to trace each request to a specific user.

With all the passwords of every user except u, the adversary not only sees all the data of the other users, he also uncovers the dummy blocks for which the ownership is limited to those users. The only outstanding blocks are those on which u holds a share (A_u). Figure 3 illustrates the distinction between various groups of blocks in the system, and also the ones used in the calculation of $\mathsf{deny}_{\mathsf{u,i}}$.

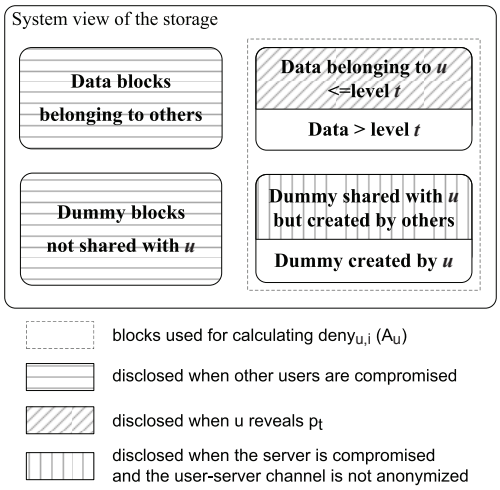


Figure 3: System view of allocated blocks

Taking into account the organization of the user data into different password levels n and P_t, operations on data blocks in level t and below are disclosed to the adversary. Let $\mathsf{op}_{u,l}^{\mathsf{data}}$ denote the number of data blocks in $D_{u,l}$, and $\mathsf{op}_u^{\mathsf{dummy}}$ denote the number of dummy blocks recorded in USB_u. The deniability of a user u (who has revealed p_t) is a function of the undisclosed blocks held by him:

$$\mathsf{deny}_{\mathsf{u,i}} = \frac{\mathsf{op}_u^{\mathsf{dummy}}}{\sum_{l>t} \mathsf{op}_{u,l}^{\mathsf{data}} + \mathsf{op}_u^{\mathsf{dummy}}} \tag{4}$$

The disclosed passwords do not affect $\mathsf{op}_u^{\mathsf{dummy}}$ in the above equation. Therefore, a bigger t improves the deniability for the data of user u being withheld from the adversary. This is intuitive, since a bigger t means that there is less user data to be hidden among the fixed pool of dummy blocks.

5.1.3 *Worst-case scenario*

When the user-server channel is not anonymized and the storage server is compromised by the adversary, the adversary is able to distinguish the creator from other share holders by monitoring the requests sent to the server. Under such a scenario, a user cannot utilize the dummy blocks that are not created by himself to provide deniability for his secret data (even if he is one of the owners of these dummy blocks). This leads to the worst-case deniability $\mathsf{deny}_{\mathsf{u,i}}$ for DRSteg since $\mathsf{op}_u^{\mathsf{dummy}}$ in Equation 4 only contains dummy blocks created by user u himself.

5.2 α-deniable DRSteg

We now show how to operationalize the dummy manipulation mechanism to secure DRSteg under the worst-case scenario described above. Specifically, we demonstrate how to manipulate dummy data to maintain the deniability above a given threshold α_T, thus making DRSteg α_T-deniable.

5.2.1 *Number of Dummy Blocks to Manipulate*

Let $\sigma_{u,l} = \mathsf{op}_{u,l}^{\mathsf{dummy}} / \mathsf{op}_{u,l}^{\mathsf{data}}$. The number of dummy blocks operated on by u, $\mathsf{op}_u^{\mathsf{dummy}} = \sum_l \mathsf{op}_{u,l}^{\mathsf{dummy}} = \sum_l \mathsf{op}_{u,l}^{\mathsf{data}} \times \sigma_{u,l}$.

Substituting into Equation (4), we have

$$\mathsf{deny}_{\mathsf{u,i}} = \frac{\sum_l (\mathsf{op}_{u,l}^{\mathsf{data}} \times \sigma_{u,l})}{\sum_{l>t} \mathsf{op}_{u,l}^{\mathsf{data}} + \sum_l (\mathsf{op}_{u,l}^{\mathsf{data}} \times \sigma_{u,l})} \quad (5)$$

In order to ensure that every $\mathsf{blk_i} \in A_u$ meets the deniability threshold of α_T no matter which password level user u chooses to surrender, we need

$$\mathsf{deny}_{\mathsf{u,i}} = \frac{\sum_l (\mathsf{op}_{u,l}^{\mathsf{data}} \times \sigma_{u,l})}{\sum_l \mathsf{op}_{u,l}^{\mathsf{data}} + \sum_l (\mathsf{op}_{u,l}^{\mathsf{data}} \times \sigma_{u,l})} > \alpha_T$$

Simplifying the above equation, we get

$$\sigma_{u,l} > \frac{\alpha_T}{1-\alpha_T} \quad (6)$$

Since $\sigma_{u,l} = \mathsf{op}_{u,l}^{\mathsf{dummy}} / \mathsf{op}_{u,l}^{\mathsf{data}}$, Equation (6) implies that to achieve the target deniability threshold α_T, the number of dummy blocks manipulated must be at least $\frac{\alpha_T}{1-\alpha_T}$ times $\mathsf{op}_{u,l}^{\mathsf{data}}$, the number of data operations.

5.2.2 Controlling dummy operations

Having determined the number of dummy blocks to manipulate, we give the procedures for controlling the dummy manipulation in DRSteg in order to achieve the deniability configured by users.

There are three types of operations on the dummy blocks – creating, deleting and voting – among which dummy creation is the easiest to control. When a user logs in at security level l, he configures σ_l (which is bigger than $\frac{\alpha_T}{1-\alpha_T}$). If x free blocks are allocated for creating or modifying a secret file, then after a random delay, the DRSteg client creates $x \cdot \sigma_l$ dummy blocks to maintain the deniability.

Deletion is more complex because a user does not know which blocks are really dummy blocks (he can only identify blocks that are not his data, as illustrated in Figure 2). To conceal the deletion of x data blocks, the DRSteg client has to delete $x \cdot \sigma_l$ dummy blocks. This is done by moving the shares of $x \cdot \sigma_l$ randomly selected blocks in $A_u - \cup_l D_{u,l}$ from USB_u to GVT after a random delay. Although some of these $x \cdot \sigma_l$ blocks may be data blocks of other users, the respective data owners will turn these (data) blocks into dummy anyway as explained next.

Now suppose that user u' logs in, and discovers that a block $\mathsf{blk} \in A_{u'}$ has been put up in GVT for deletion. If blk does not contain his data, i.e., if $\mathsf{blk} \in (A_{u'} - \cup_l D_{u',l})$, u' will support the deletion by adding his votes on blk in GVT. If blk is a data block of u' (i.e., $\mathsf{blk} \in \cup_l D_{u,l}$), then u' has to migrate the content to a new block before voting for the deletion. As discussed in Section 4.2, this is to avoid leaving clues that blk contains user data.

5.2.3 Security Discussions

There are several security concerns relating to dummy manipulation. First, in our current design, every block operation is either a direct data operation or the effect of a data operation. Besides introducing random delays, their association could be masked by breaking each of the dummy creations and block deletions into smaller steps and interleaving them with data block operations. In addition, DRSteg could initiate dummy operations independently of data operations. These enhancements will be incorporated in future work.

Second, the parameter $\sigma_{u,l}$ is of special interest to the adversary, who might force the victims to reveal their choices of $\sigma_{u,l}$. With the $\sigma_{u,l}$ values, the adversary may estimate the actual number of data block operations, thus limiting the victims' flexibility to attribute as dummy those data blocks that they are trying to hide. To substantiate his denial in the event of an attack, DRSteg furnishes each user u with a fake $\sigma_{u,t}^{\mathsf{fake}}$ at log-out, where t is the password level that the user is willing to disclose. $\sigma_{u,t}^{\mathsf{fake}}$ is calculated as the ratio between the number of blocks claimed to be dummy (including dummy blocks and hidden data blocks), and the number of revealed data blocks: $\sigma_{u,t}^{\mathsf{fake}} = (\Sigma_{l>t} \mathsf{op}_{u,l}^{\mathsf{data}} + \mathsf{op}_u^{\mathsf{dummy}}) / \Sigma_{l \le t} \mathsf{op}_{u,l}^{\mathsf{data}}$.

Another potential security threat is, if the adversary is able to take snapshots of the storage content with infinitesimal delay, he may be able to distinguish dummy blocks from data blocks. Troncoso et al. [18] showed that this distinction is possible because data blocks belonging to the same file are often accessed one after another, whereas dummy blocks are accessed individually and are not likely to exhibit the same access pattern. To mitigate against such a threat, one possible solution is to introduce dummy files into DRSteg. A dummy file would span several dummy blocks, which are then accessed sequentially like data blocks. In order to present similar access pattern as data files, dummy files should also be accessed frequently. Such an improvement in dummy file operations is left for future work.

6. EVALUATION

6.1 Empirical Evaluation on Deniability

To investigate DRSteg's ability to maintain user-specified deniability thresholds under multiple-snapshot attacks, we perform an empirical evaluation by re-playing file operations logged in a typical office environment. We deployed a logger to record the file operations (operation type and time) on the computers of 12 graduate students in our lab. Over 9 days, we recorded more than 50,000 *user* file operations[5].

We begin by mirroring the user files of all 12 computers in DRSteg, which add up to about 1 Tbyte of data. We also initialize the same number of dummy blocks, making the original utilization of data blocks 0.5. The shares for data and dummy blocks are distributed randomly among the 12 users. We assume that users are automatically logged out from the stegfs system after some period of inactivity (10 minutes in our experiments), and they login again right before their next observed data operations. For each session, the user enters the password to one of his security levels l (randomly chosen by our simulator) and picks a $\sigma_{u,l}$ value (chosen to follow a power-law distribution $p(\sigma) \propto L(\sigma)\sigma^{-\xi}$ assuming that more users will tend to choose lower σ values to minimize overhead). We set $\alpha_T = 0.4$, $\sigma_{min} = 0.7$ and $\xi = 3.0$ for all users. The parameters and statistics are summarized in Table 3.

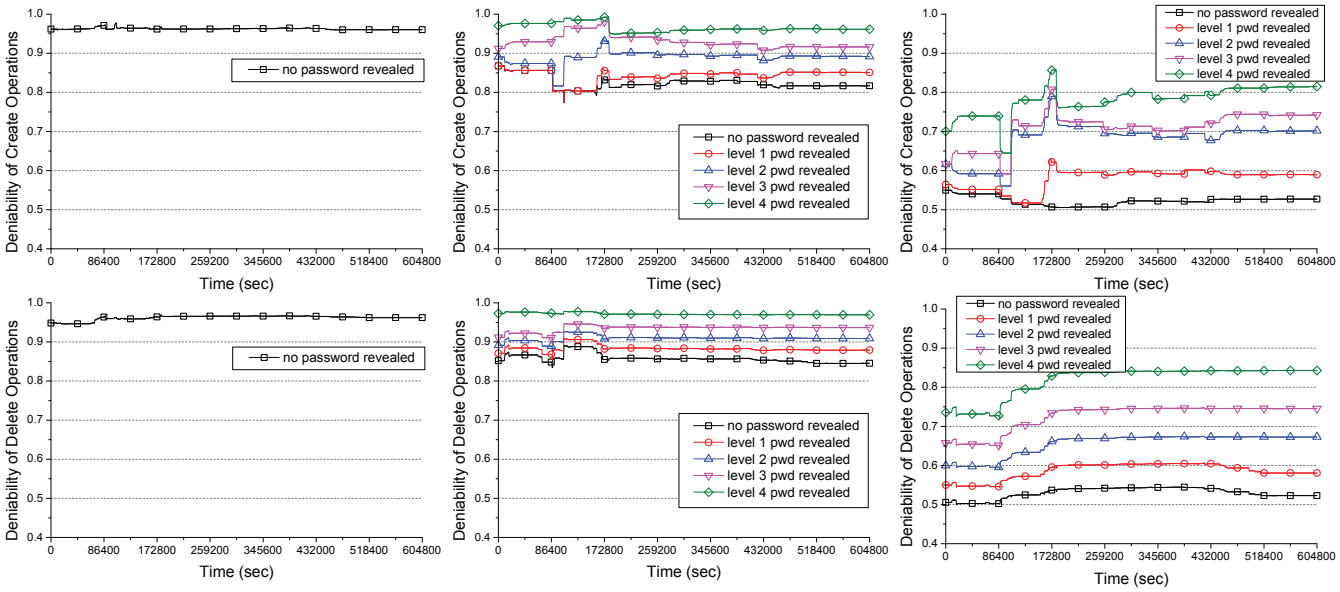
We use the first two days of logs to warm up DRSteg. As the remaining seven days of traces are executed, we take a snapshot of the disk image every 10 minutes. Figure 4 shows the deniability for one of the (randomly chosen) users by comparing each successive snapshot with the first one.

Figure 4(a) shows the deniability under the *passive-adversary* scenario, calculated with Equation 3. The upper graph gives

[5]We assume that the operating system and software programs are not installed in the stegfs partition.

Parameter	Value	User-log Statistics	Value	Simulated DRSteg Statistics	Value
# of users	12	Total logging time	9 days	Initial amt. of data blocks	1011.34 GB
α_T	0.4	# of file operations	50,113	Initial amt. of allocated blocks	2022.68 GB
# of security levels	5	Data blocks created	26.613 GB	# of user sessions	294
Interval before auto logout	10 mins	Data blocks deleted	80.069 GB	Final amt. of data blocks	970.60 GB
Avg. # of shares per block	3	Data blocks modified	160.317 GB	Final amt. of allocated blocks	1995.89 GB

Table 3: Simulation parameters and statistics



(a) Passive-adversary scenario (b) Anonymous-channel scenario (c) Worse-case scenario

Figure 4: Deniability of DRSteg under different scenarios

the deniability with respect to block creation evidence, while the lower is for delete operations. As seen from the graphs, sharing dummy blocks among users enables individual users to enjoy high deniability.

Next, we examine Figure 4(b) for the *anonymous-channel* scenario, which is calculated with Equation 4. Here, the selected user has revealed up to level t of his passwords (the lines in the graphs represent different settings of t), whereas the other users have revealed all their passwords. Since the selected user can only rely on the operations on dummy blocks which are recorded in his UMB, the deniability is lower than that in the previous scenario. Nevertheless, DRSteg still manages to achieve high deniability.

Turning to the *worst-case* scenario where the adversary is aware of the creator of every block, Figure 4(c) shows the deniability levels achieved. In this scenario, deniability is derived solely from operations on the dummy data created by the user himself, which explains the much reduced deniability. Even so, DRSteg manages to keep the deniability above the configured threshold of $\alpha_T = 0.4$.

The deniability for the other 11 users are similar to the results in Figure 4 quantitatively and qualitatively. In particular, the lowest deniability observed for the worst-case attack scenario is 0.46. These results affirm the security property of our proposed DRSteg.

6.2 Implementation and Performance Evaluation

We have implemented DRSteg as a file system module in parallel with ext3 in Linux kernel 2.6, on the client machines which communicate with the shared storage through a server (see Figure 1). The client module manages the blocks in the shared storage automatically according to the password entered by the user. This includes creating new data and dummy blocks (and allocating shares to other owners), voting blocks for deallocation, etc. We explain below how the storage is organized by the system and benchmark the performance of DRSteg.

6.2.1 *File system construction*

In our DRSteg file system, the (remote) disk storage is partitioned into blocks of 1 Kbyte in size by default. A bitmap tracks the allocation status of the blocks: 1 corresponds to an allocated block and 0 a free block. An allocated block is either a dummy or a data block, both of which appear to contain random patterns.

To accelerate access to directories and files, DRSteg uses a designated storage area, called the *super block* (see Figure 5), to store inode structures so that they can be located efficiently. The super block is essentially a mini-DRSteg system for the addresses of inode roots, and is calved into fixed-size slots that are capable of holding one address each. A slot may be a free slot, a dummy slot, or may contain

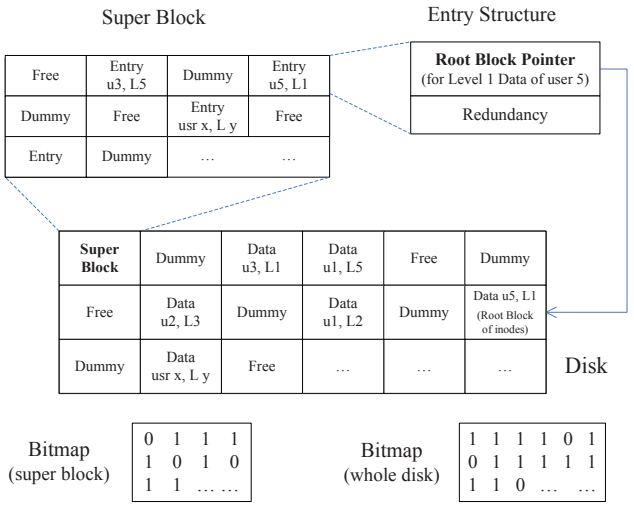


Figure 5: Block organization in DRSteg

the encrypted address of an inode root (with redundancy so that it is distinguishable from random bits upon decryption). Each password level of a user is allocated one slot. Since the super block is expected to be only a few Kbytes in size, it can be scanned quickly to find the inode roots for each user. The super block has its own bitmap to track slot allocation, while it shares the same set of user share boxes and the global voting table with the main file system.

6.2.2 *Performance Evaluation*

The key parameters of the computing hardware for our experiments are listed in Table 4, while Table 5 summarizes the workload parameters and their default settings.

The first experiment is designed to study how well DRSteg performs. For comparison, we include StegCover, StegRand [4] and NSteg [16] as baselines. StegCover is configured with 20 cover files (the authors recommended 16 to 100 [4]). For StegRand, we use a replication factor of 4 to reduce the probability of data loss [15]. NSteg is set to populate 30% of the disk with dummy blocks during initialization. We also include two settings of the native Linux file system (ext3) in our tests. In the CleanDisk setting, data files are loaded into a freshly formatted native Linux partition, so that the files occupy contiguous disk blocks; with file operations translating to sequential I/Os, CleanDisk gives the best-case timings. In contrast, results of FragDisk are obtained with a well-used ext3 partition in which the free space is fragmented.

In the first experiment, we configure DRSteg with $\sigma_{u,l} = 0.25$, which produces a worst-case deniability of 0.2. For a given concurrency level, we generate file creation requests one after another for each user and measure the elapse time. Figure 6(a) shows the average write time for various file systems, with the number of concurrent users ranging from 1 to 32. Every performance result is averaged over 1000 observations.

The results show that StegCover is the worst performer; this is because each file operation translates into disk I/Os on several cover files. StegRand is also slow because it has to modify all the replicas. DRSteg and NSteg use a bitmap to track the status of disk blocks, so they can ensure data integrity with just one copy of each data file. Consequently, they are substantially faster than StegCover and StegRand. They are slower than FragDisk though, because they encrypt the protected files block by block and spread them across the disk, resulting in higher fragmentation.

Parameter	Value
CPU	Intel Duo Core 2.53GHz
RAM	2GB (1GB DDR2-667 x 2)
Hard Disk	SATA 7200rpm, 250 GB with 8MB cache

Table 4: Hardware Parameters

Parameter	Default Value
Capacity of the test partition	40 Gbytes
Size of each disk block	1 Kbytes
Number of blocks for each file	1024
File access pattern	Interleaved

Table 5: Workload Parameters

Recall that DRSteg needs to write additional messages into the UMB's during block creation and generate dummy operations dynamically. As the file creation requests in our experiment are issued one after another with no delay, the file system is fully loaded, leaving no idle period for DRSteg to schedule its dummy operations. Thus, the dummy operations add directly to the write times, and the observed timings represent the worst-case performance of DRSteg. For example, with $\sigma_{u,l} = 0.25$ it is roughly 30% slower than NSteg. This is the cost paid by DRSteg to achieve better security protection, compared to NSteg which is not able to relocate its dummy blocks.

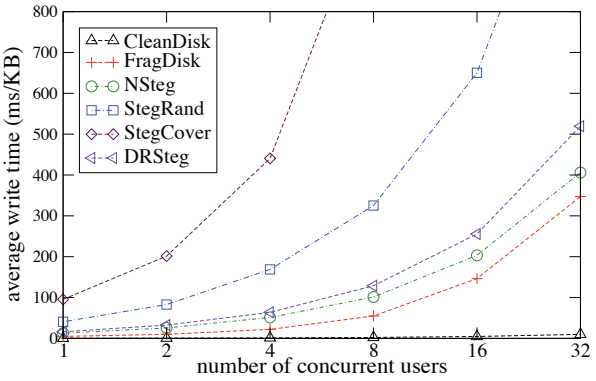
In the second experiment, we investigate the performance of DRSteg under different load conditions. The load condition is determined by various factors, including the σ parameter that controls the amount of dummy operations, the concurrency level, and the activity level of each user. We model the activity level after a Poisson process with mean arrival rate of λ block operations per minute. The results are summarized in Figure 6(b), which plots the average write time against λ for several σ-concurrency combinations.

We first consider the impact of λ. For every σ-concurrency combination, DRSteg's write time is short initially because there are ample lull periods during which dummy operations can be scheduled so as to reduce contention with data operations. Such opportunities diminish with increasing λ, leading to longer write times observed in the figure. Next, we compare the three σ-concurrency combinations with $\sigma = 0.25$. With the same σ and λ settings, raising the concurrency level introduces more contention between the data and dummy operations and lengthens the write time. Similarly, a bigger σ generates more dummy operations to cover the data operations, again resulting in longer write times.

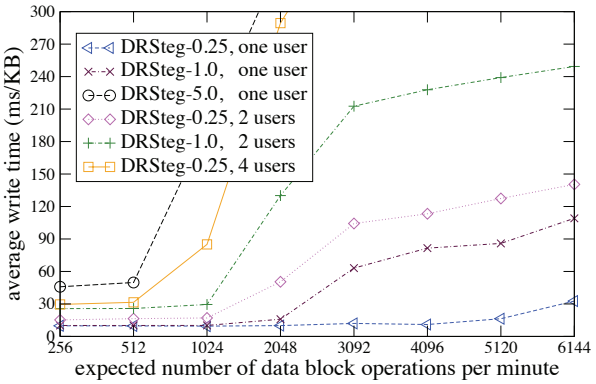
In summary, our experiment demonstrates that DRSteg is capable of striking a wide range of trade-offs between deniability and system performance. If high deniability is required, the file system should be configured with enough resources to prevent it from becoming overloaded. On the other hand, to support a heavy workload, we could configure DRSteg for a lower deniability assurance.

7. CONCLUSION

In this paper, we address the threat to steganographic



(a) Comparing with previous stegfs designs



(b) Trade-off between deniability and performance

Figure 6: Performance evaluation results

file systems (stegfs) that arises when the underlying storage is untrusted and shared by multiple users. In such systems, an adversary could obtain and analyze multiple snapshots of the storage content to deduce the existence of secret user data. To counter the threat, we introduce a Dummy-Relocatable Steganographic (DRSteg) file system that employs novel techniques to share and relocate dummy data at runtime. This enables users to surrender only some of their data, and attribute any unexplained changes across snapshots to dummy operations. The deniability enjoyed by users is configurable individually. DRSteg guarantees the integrity of the protected data, except where users voluntarily overwrite data under duress. A trace-driven simulation confirms the security of our scheme. Further experiments on a Linux prototype demonstrate that DRSteg is able to effectively trade off deniability with system performance.

8. REFERENCES

[1] eCryptfs, a POSIX-compliant enterprise-class stacked cryptographic filesystem for Linux. `https://launchpad.net/ecryptfs`.

[2] Encrypting File System in Windows XP and Windows Server 2003. `http://www.microsoft.com/technet/prodtechnol/winxppro/deploy/cryptfs.mspx`.

[3] R. J. Anderson and E. Biham. Two practical and provably secure block ciphers: Bears and lion. In *Proceedings of the Third International Workshop on Fast Software Encryption*, pages 113–120, 1996.

[4] R. J. Anderson, R. M. Needham, and A. Shamir. The steganographic file system. In *Proceedings of the 2nd International Workshop on Information Hiding*, pages 73–82, 1998.

[5] M. Blaze. A cryptographic file system for unix. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 9–16, 1993.

[6] R. Canetti, C. Dwork, M. Naor, and R. Ostrovsky. Deniable encryption. In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science*, pages 90–104, 1997.

[7] G. Cattaneo, L. Catuogno, A. D. Sorbo, and P. Persiano. The design and implementation of a transparent cryptographic file system for unix. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 199–212, 2001.

[8] C. Diaz, C. Troncoso, and B. Preneel. A framework for the analysis of mix-based steganographic file systems. In *Proceedings of the 13th European Symposium on Research in Computer Security*, pages 428–445, 2008.

[9] J. Domingo-Ferrer and M. Bras-Amorós. A shared steganographic file system with error correction. In *Proceedings of the 5th International Conference on Modeling Decisions for Artificial Intelligence*, pages 227–238, 2008.

[10] C. Giefer and J. Letchner. Mojitos: A distributed steganographic file system. Technical report, Univerisity of Washington, 2004.

[11] F. Graf and S. D. Wolthusen. A capability-based transparent cryptographic file system. In *Proceedings of the 2005 International Conference on Cyberworlds*, pages 101–108, 2005.

[12] S. Hand and T. Roscoe. Mnemosyne: Peer-to-peer steganographic storage. In *Proceedings of the First International Workshop on Peer-to-Peer Systems*, pages 130–140, 2002.

[13] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings of the Annual IEEE Symposium on Foundations of Computer Science*, pages 364–373, 1997.

[14] L. Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11), 1981.

[15] A. D. McDonald and M. G. Kuhn. StegFS: A steganographic file system for Linux. In *Proceedings of the 3rd International Workshop on Information Hiding*, pages 462–477, 2000.

[16] H. Pang, K.-L. Tan, and X. Zhou. StegFS: A steganographic file system. In *Proceedings of the 19th International Conference on Data Engineering*, pages 657–668, 2003.

[17] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 199–212, 2009.

[18] C. Troncoso, C. Diaz, O. Dunkelman, and B. Preneel. Traffic analysis attacks on a continuously-observable steganographic file system. In *Proceedings of the 9th Information Hiding*, pages 220–236, 2008.

[19] C. P. Wright, M. C. Martino, and E. Zadok. NCryptfs: A secure and convenient cryptographic file system. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 197–210, 2003.

[20] X. Zhou, H. Pang, and K.-L. Tan. Hiding data accesses in steganographic file system. In *Proceedings of the 20th International Conference on Data Engineering*, pages 572–583, 2004.

Heap Taichi: Exploiting Memory Allocation Granularity in Heap-Spraying Attacks

Yu Ding[1,2], Tao Wei[1,2]*, TieLei Wang[1,2], Zhenkai Liang[3], Wei Zou[1,2],

[1]Institute of Computer Science and Technology, Peking University
[2]Key Laboratory of Network and Software Security Assurance(Peking University), Ministry of Education, Beijing 100871, China
[3]Department of Computer Science, School of Computing, National University of Singapore

ABSTRACT

Heap spraying is an attack technique commonly used in hijacking browsers to download and execute malicious code. In this attack, attackers first fill a large portion of the victim process's heap with malicious code. Then they exploit a vulnerability to redirect the victim process's control to attackers' code on the heap. Because the location of the injected code is not exactly predictable, traditional heap-spraying attacks need to inject a huge amount of executable code to increase the chance of success. Injected executable code usually includes lots of NOP-like instructions leading to attackers' shellcode. Targeting this attack characteristic, previous solutions detect heap-spraying attacks by searching for the existence of such large amount of NOP sled and other shellcode.

In this paper, we analyze the implication of modern operating systems' memory allocation granularity and present Heap Taichi, a new heap spraying technique exploiting the weakness in memory alignment. We describe four new heap object structures that can evade existing detection tools, as well as proof-of-concept heap-spraying code implementing our technique. Our research reveals that a large amount of NOP sleds is not necessary for a reliable heap-spraying attack. In our experiments, we showed that our heap-spraying attacks are a realistic threat by evading existing detection mechanisms. To detect and prevent the new heap-spraying attacks, we propose enhancement to existing approaches and propose to use finer memory allocation granularity at memory managers of all levels. We also studied the impact of our solution on system performance.

1. INTRODUCTION

Heap spraying is a new attack technique commonly used in recent attacks to web browsers [4–8, 36]. In a heap-spraying attack, attackers allocate objects containing their malicious code in the victim process's heap, and then trigger a vulnerability to force the victim process to execute code from the heap region. Compared to traditional buffer overflow attacks, heap spraying is simpler, as there is no need to know the detailed layout of memory regions surrounding the buffer vulnerable to overflow, but the heap object locations are not predictable. In order to increase the chance of success, existing heap-spraying techniques allocate lots of heap blocks filled with a large amount NOP-like instructions (e.g. `0x90`, `0x0c 0x0a`), called *NOP sled*, and followed by the malicious shellcode. The NOP sled serves as the *landing area* of the shellcode, which leads the execution to the shellcode if the victim process jumps to anywhere in the NOP sled.

Since a typical heap object used in a heap-spraying attack is in the form of "NOP sled + shellcode," the *large amount of NOP sled* and *existence of shellcode* are the main characters used by heap-spraying attack detectors. Accordingly, existing approaches to detect heap-spraying attacks mainly fall into two types: *sled-oriented* and *shellcode-oriented*. Shellcode-oriented methods detect heap-spraying attacks by detecting the existence of shellcode. For example, Egele et al. [21] detect heap-spraying attacks by inspecting the JavaScript string objects to identify shellcode using lightweight emulation [9]. However, this type of approach have difficulty in dealing with shellcode obfuscation techniques, such as, shellcode encoding [28, 34], encryption [46], polymorphism [20, 24], and other obfuscation schemes [17, 27, 31, 37].

A more successful type of techniques to detect heap-spraying attacks are sled-oriented [11, 29, 32, 42]. Such techniques focus on identifying large chunks of NOP sled. In particular, NOZZLE [32] uses static analysis to build the control-flow graph (CFG) of heap memory blocks and measures the size of NOP sled, called *surface area*, across a process's entire heap region. If the percentage of surface area is above a certain threshold, NOZZLE reports an attack. NOZZLE assumes that heap-spraying attacks must inject a large number of executable codes (especially NOP sled) because attackers cannot predict the location of their malicious code. Next, we will show that this assumption is not always valid.

We observe that modern operating system memory allocation behavior is more predictable than we usually believe, even in the presence of address space layout randomization (ASLR). For instance, the Windows-family systems (from Windows XP to Windows 7) enforce a memory allocation granularity of 64K bytes [22, 33], which makes all memory blocks *directly* allocated by Windows (using API `VirtualAlloc`) aligned to a 64K-byte boundary. As a result, addresses of such heap blocks are less random. For example, a particular address in a 1MB block only has 16 possible locations, much less than the one million possible locations if the heap block can be allocated at random addresses. We discuss this in detail in Section 3.

*Corresponding author. Email: *weitao@icst.pku.edu.cn*

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

A new attack. Based on the above analysis, we present a new heap-spraying technique, called *Heap Taichi*, which can evade existing detection mechanisms. By precisely manipulating the heap layout, Heap Taichi only needs to put executable code at a small number of offsets in a heap block, and thus makes the "large of NOP sled" feature in traditional heap-spraying attacks unnecessary.

To demonstrate the feasibility of Heap Taichi, we made proof-of-concept heap-spraying attacks using Heap Taichi. Our experiments showed that the surface area of a Heap Taichi attack is significantly less than the acceptable threshold used in existing solutions. We also studied the impact of different memory-allocation granularity on heap-spraying attacks and system performance, and found that larger memory allocation granularity gives attackers more flexibility without significant gain in performance.

To address this problem, we proposed methods to enhance existing heap-spraying attack detection techniques by considering memory allocation granularity, and experimented with new ways of memory allocation.

Contributions:

- We analyze the implication of modern operating systems' memory allocation granularity on heap-spraying attacks, and present a new heap-spraying technique utilizing the weakness of memory alignments, which can effectively evade existing detection tools.
- We present four heap object structures that do not require a large amount of NOP sled. We provide insight into the relationship between memory alignment size and heap-spraying attack surface areas.
- We implement proof-of-concept Heap Taichi, and measure the attack surface areas of these attacks. Experiments showed that our heap-spraying attacks are a realistic threat, which can evade existing detection tools.

2. HEAP SPRAYING AND DEFENSE

In this section, we describe a typical heap-spraying attack, and discuss existing defense mechanisms.

2.1 Heap-spraying attacks

Throughout the paper, we use the term *heap region* to refer to all the memory areas of a process's heap. We use the term *heap block* to refer to the memory block allocated for heap, e.g., the blocks allocated by Windows's memory management through the `VirtualAlloc` family APIs. We call individual objects allocated on the heap *heap objects*, e.g., objects allocated by the API `HeapAlloc`. Therefore, a heap region consists of several heap blocks, and a heap block contains one or more heap objects.

Figure 1 illustrates a typical heap-spraying attack found by our web crawler. The attack is launched by malicious JavaScript in a web page, targeting a vulnerability in the Internet Explorer version 6 or version 7 [18]. In the first step of this attack, attackers create a large amount of heap objects. Each heap object is filled with a large number of NOP-like instructions (`0x0c0c`, the instruction `or al, 0ch`) followed by a block of malicious shellcode. Illustrated in the right-hand side of Figure 1, the large white areas are the NOP-like instructions, while the grey areas are the shellcode. If attackers can hijack the process's execution to any byte in the range of NOP-like instructions, the malicious shellcode will be executed. Although attackers cannot know the exact address of the injected code, when the browser process's heap region is very large, certain

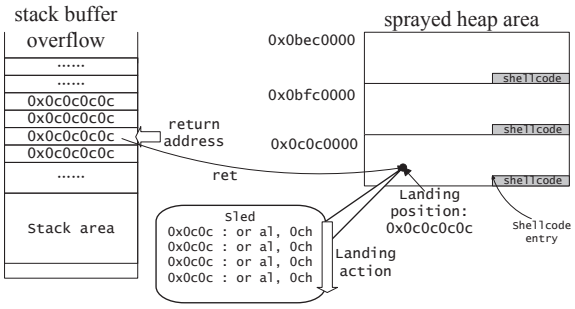


Figure 1: The traditional heap spraying with stack buffer overflow exploit.

range of address, such as `0x0c0c0c0c`, will be in the region of allocated heap objects, as is illustrated in Figure 1.

After the heap is prepared with malicious shellcode, the second step of this attack is to exploit a vulnerability in the victim process, forcing the victim process to transfer control to the sprayed heap region. Any vulnerability that can be exploited to affect the control flow can be used in this step. Here we show an example using a stack-based buffer overflow vulnerability, illustrated in the left-hand side of Figure 1. By exploiting the buffer overflow vulnerability, attackers inject lots of `0x0c` characters onto the stack, overwriting the return address. When the program returns using the corrupted return address, its execution is redirected to the address `0x0c0c0c0c`, which is in the NOP sled of a sprayed heap object. The victim program will continue through the NOP sled and reach attackers' shellcode.

Thanks to the defense mechanisms against buffer overflow attacks, it is very hard for attackers to know the exact address of their shellcode. Therefore, they cannot use the stack overflow to execute their shellcode directly. In contrast, heap-spraying attacks do not require attackers to know the detailed layout of the data structures of the targeted memory region. But the object addresses on the heap are harder to predict, even with the deployment of ASLR [22, 33, 40, 44]. To increase the chance of success in the second step of the attack, attackers usually put a long NOP sled before the shellcode in their heap objects, and they have to inject a large amount of heap objects containing shellcode, so that the jump target of the attack will be covered by injected code with a high probability. Otherwise, if the victim process jumps into the middle of shellcode, or even jumps out of the heap region sprayed by the attacker, the victim process often crashes because of invalid memory access or invalid instructions.

In the rest of the paper, we use the following terms to describe the behavior of a heap-spraying attack. 1) We call the execution after the exploit and before running the shellcode a *landing action*. In traditional heap-spraying attacks, the landing action usually runs on the huge sled area, byte by byte. The traditional sled is filled with NOP like bytes, such as `0x90` (NOP), `0x0c0c` (or al, 0ch) and these bytes lead to smooth landing actions. On the contrary, landing actions executing some jump instructions, such as `jmp`, are called bumpy landing. 2) The place where the landing action starts is called *landing position*, or *landing point*. 3) The notion of *surface area* is defined in the NOZZLE paper [32] as the number of available landing positions in one heap object. 4) The *normalized attack surface area (NSA)* is a heap object's surface area divided by the heap object's size. The normalized attack surface area represents the percentage of sled in a memory block. It also represents the possibility of successfully executing the shellcode when execu-

tion randomly falls into a heap object. 5) The *shellcode entry* is the starting point of the shellcode.

2.2 Existing defense mechanisms

Existing defense mechanisms against heap-spraying attacks can be classified into two main types based on the analysis they perform on heap objects. Approaches of the first type detect shellcode by searching for common patterns of shellcode. Approaches of the second type analyze the control flow structure of heap objects to identify common structures used in heap-spraying attacks.

Egele et al. [21] is an example of the first type. It monitors all strings objects allocated in a browser's JavaScript engine, and reports an attack when there is shellcode detected in string objects created by the script. To detect shellcode, it uses the libemu library to identify suspicious and valid instruction sequences longer than 32 bytes. As is discussed in the paper [21], attackers can evade detection by breaking shellcode into multiple fragments smaller than 32 bytes and linking them with indirect jump/call instructions. However, unless attackers can precisely control the landing position, they still need a large portion of NOP sled to make a reliable attack, which is well over the 32 byte threshold.

NOZZLE [32] is an example of the second type. Given a heap object, it disassembles possible x86 instructions in the object and build a control flow graph (CFG). As we have described earlier in this section, the heap block used in a typical heap-spraying attack contains a block of shellcode, and the rest of the heap block contains instructions leading to the shellcode. NOZZLE searches for this property in the CFG by identifying the location S that can be reaching from most of other locations in the heap object. The total number of locations that lead to S is the *surface area* of the heap object. In other words, NOZZLE draws the CFG of the heap block. For each basic block in the CFG, it counts the number of instructions that connect to the basic block. NOZZLE then calculates the surface area of the entire heap. When the surface-area-to-heap-size ratio is greater than a threshold, NOZZLE reports a heap-spraying attack. This approach is more accurate than the first type approaches, because it looks for more intrinsic properties of heap-spraying attacks: when the location of shellcode is not predictable, it is necessary to include large surface areas to increase the chance for success.

Both types of existing solutions assume *attackers have little information about the address of their shellcode*. With this assumption, attackers cannot break sled and shellcode into small pieces to evade the approach of Egele et al.; they also cannot evade NOZZLE by only including very little NOP sled instructions. This assumption is valid if heap objects are allocated randomly without restriction. However, the randomness of heap object allocation is limited by memory alignment enforced in operating systems. Next, we discuss its impact on heap memory allocation and describe an attack that can evade both types of defense mechanisms.

3. HEAP SPRAYING WITH LITTLE SURFACE AREAS

Memory alignment is commonly adopted in modern operating systems for better memory performance. With memory alignment, a memory block allocated for a process cannot start from arbitrary addresses. Instead, the addresses must be multiples of the alignment size defined by the system.

In this section, we analyze the memory allocation behavior of the Windows platform and its implication on heap-spraying attacks. Then we describe a new attack technique that can evade existing heap-spraying detection mechanisms. Note that other operating systems such as Linux have a similar memory allocation behavior to Windows, which differs mainly in the default memory alignment size.

3.1 Windows memory allocation granularity

Windows memory alignment is controlled by the *allocation granularity*. On all existing Windows platforms, the value of allocation granularity[1] is always 64K [33]. This size 64K was chosen in part for supporting future processors with large page sizes [22], as well as solving relocation problems on existing processors [3]. The memory allocation granularity only affects user-mode code; kernel-mode code can allocate memory at the granularity of a single page [22]. As a result of the Windows memory allocation granularity, almost all of the base addresses of non-free regions are aligned with 64K boundaries. In a process's memory space, only few regions (allocated by kernel-mode code [33]) are not aligned. Even with ASLR enabled [40], the alignment of memory region addresses is not affected. On Linux systems, the memory allocation granularity is 4K bytes.

Therefore, taking Windows as an example, all heap blocks allocated by user-mode code start from 64K boundaries. Note that heap objects allocated by `HeapAlloc` can still start at random addresses in a heap block, but we have an interesting observation: when a heap object is bigger than a certain threshold, 512K in our experiment, Windows always allocates a separate heap block for this object. That is, the addresses of large heap objects are also aligned according to the allocation granularity, thus more predictable.

What is the implication of such a memory allocation behavior on heap-spraying attacks? Recall that in the second step of a heap-spraying attack, after the attacker triggers a control-hijacking exploit successfully, the victim process's EIP register is loaded with a value assigned by the attacker. If the starting addresses of heap objects are fully random, the EIP can fall anywhere in a heap object. For example, when the heap object's size is 512K bytes, the hijacked EIP can point to any byte of the 512K bytes. This is the main reason for requiring a large amount of NOP-like instructions in heap-spraying attacks. However, the Windows memory allocation granularity makes large heap objects' addresses much more predictable. If an EIP assigned by an attacker have few possible locations in a large heap object, the attacker only need to put jump-equivalent instructions at those locations to guide the victim process to execute malicious shell code, which breaks the assumptions, relied on by previous defense mechanisms. As a result, the large block of NOP sled is no longer necessary for a heap-spraying attack with high chances to succeed.

In fact, an EIP assigned by an attacker can only point to *EIGHT* possible locations in a 512K-byte heap object, which is explained in Figure 2 using the address `0x0c0c0c0c`. Due to the 64K (0x10000) Windows memory allocation granularity, a 512K-byte heap object covering the address `0x0c0c0c0c` can only start from `0x0c050000`, `0x0c060000`, ..., `0x0c0c0000`. Therefore, the offset of the address `0x0c0c0c0c0` inside the object have eight possible values: `0x70c0c`, `0x60c0c`, ..., `0x00c0c`. On each of these offsets, if the attacker puts a few bytes, say 20 bytes (the unconditional jump instruction takes five bytes on 32-bit x86), of jumping instructions, the resulting surface area is very little: 160 bytes out of a 512K-byte object.

[1]It can be retrieved by the `GetSystemInfo` API (the `dwAllocationGranularity` member of the returned `SYSTEM_INFO` structure).

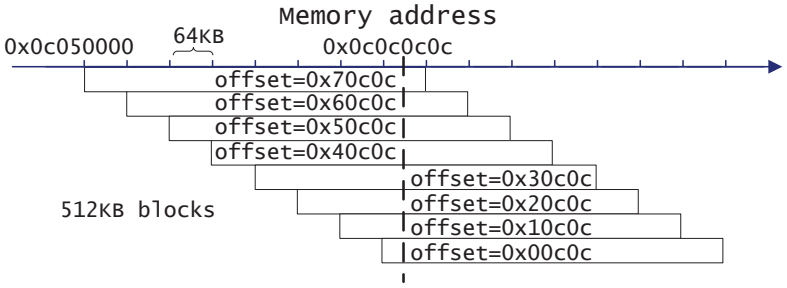


Figure 2: Possible offset of `0x0c0c0c0c` in a 512KB heap blocks.

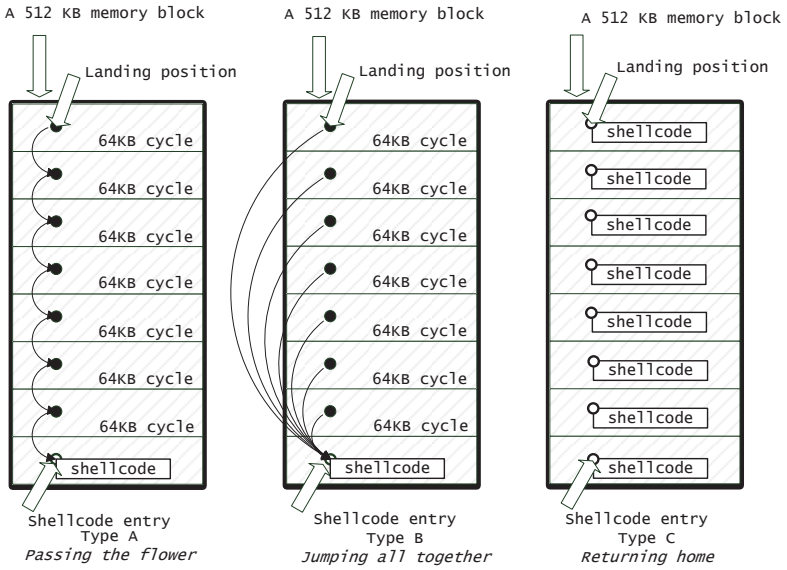


Figure 3: Structures of new heap-spraying memory blocks.

3.2 Structure of malicious heap objects with little surface area

As is discussed in the previous section, given a specific address $addr$ in the heap region, the landing action can only start at few offsets in a large heap object. Executable code at other offsets will never be the direct jump target when the process transfers control to the address $addr$. With this new insight, we describe a few new structures of malicious heap objects that result in very little surface area.

The general idea is to put jump-equivalent instructions at possible landing positions to guide execution into attackers' shellcode. The shellcode is a small piece of code connected by jump-family instructions, which can evade the approaches that detect valid instruction sequences. Figure 3 shows three types of the new heap block structures that have little surface area. In this figure, each rectangle with bold boundary stands for a heap object. The shadow areas are bytes with random values. The possible landing positions are represented as solid dots. Shellcode is represented as white rectangles, with a circle indicating its entry point.

- In the Type A structure, attackers first copy the block of malicious shellcode into the heap object. The landing positions are chained together to reach the shellcode entry. That is, each landing position is a set of jump-equivalent instructions that point to the next landing position. The instructions at the last landing position lead to the shellcode entry.
- In the Type B structure, attackers put jump-equivalent instructions at the possible landing positions. Each group of jump instruction will jump to the shellcode entry.
- In the Type C structure, the malicious shellcode is directly put at each landing position. By using this kind of memory blocks, the landing action is eliminated and the shellcode is executed immediately after the exploit is triggered.

In the Type C structure, although there are several copies of shellcode, the surface area is as small as one copy because the copies of shellcode are not connected. The Type C structure requires the shellcode size to be smaller than the alignment granularity. To launch such an attack on an operating system using a small alignment granularity, say 32 bytes, we introduce the Type D heap object structure, which is an improved Type C structure.

Shown in Figure 4, the main idea of this structure is that we can split the shellcode into pieces and link these pieces with jump instructions. We place jump instructions at each landing point to jump to the shellcode. Similar to the Type C structure, although there are lots of shellcode copies in the heap block, the measurable surface area is small. We illustrate this type of structure by an example. Assuming the memory allocation granularity is 32 bytes, we construct a 512K-byte heap block using the Type D structure, which includes 1024 copies of shellcode. In the heap block, we need to create $512K/32 = 16384$ landing points. Each landing point connects to one of the shellcode copies sequentially or arbi-

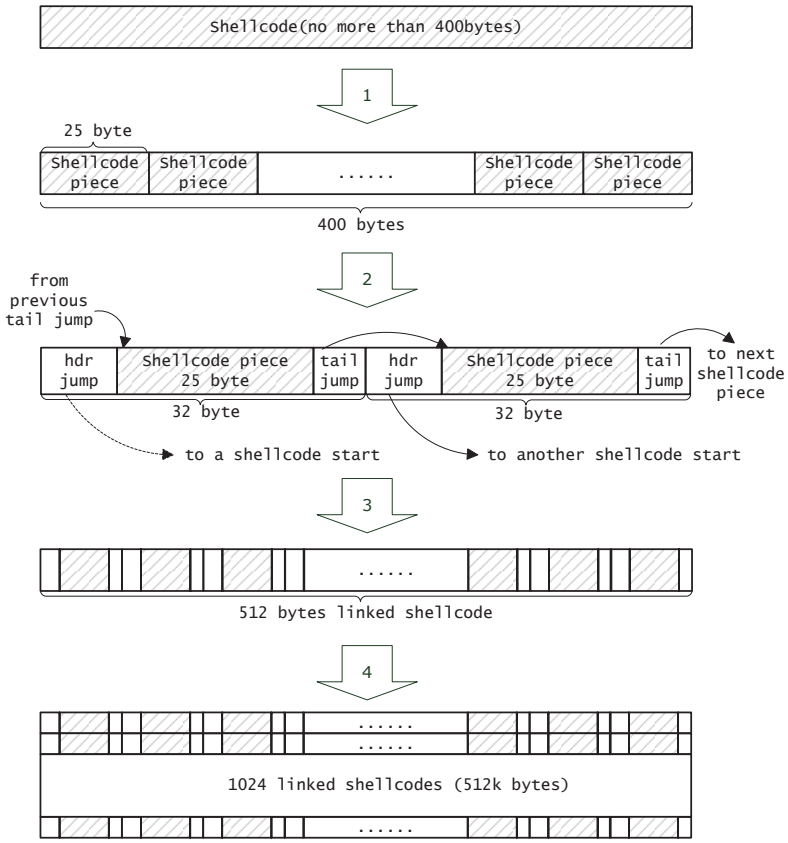


Figure 4: Type D layout 'Dropping around'

trarily. This transformation is still a "sled construction" technique, which plants landing points inside the shellcode. The shellcode features are not changed after these landing points inserted.

Type D structures can be created using the following technique. Given a piece of shellcode[2], we first split it into pieces, where each piece is less than or equal to 25 bytes. If a piece is less than 25 bytes, we append a few NOP-like instructions to it to make the size of all pieces 25 bytes. To connect the shellcode pieces, we enclose each shellcode piece between a prologue and an epilogue, shown in Step 2 of the figure. The prologue is called "*header (hdr) jump*" and it's a jump instruction (5 bytes, jump near, relative, displacement relative to next instruction) pointing to the shellcode's starting position. We need to distribute the header jumps to the start of 1024 copies of shellcode evenly. In the attack, the prologues are put at landing points. The epilogue is called "*tail jump*" and it's a jump instruction (2 bytes, jump short, relative, displacement relative to next instruction). In the attack, the epilogues connect the shellcode pieces. The tail jump only jumps $2 + 5 = 7$ bytes forward. So with the prologue and epilogue, each shellcode piece is extended to $25 + 5 + 2 = 32$ bytes. In the third step, we combine 16 such 32-byte pieces to form shellcode of $16 \times 32 = 512$ bytes. We call it a *512-byte linked shellcode*. To fit the selected original shellcode into such a block, the shellcode size should be less than $25 \times 16 = 400$ bytes. Finally, we merge 1024 linked shellcode pieces into one heap memory block. There are $1024 \times 16 = 16384$ header jumps inside the heap memory block and they are the landing positions.

The final heap memory block will be used in our new heap-spraying attack. The possible landing positions are at each 32 byte boundary. So we could exploit to address such as `0x0c0c0c20`, `0x0c0c0c40`, `0x0c0c0c60`, and etc. When the execution starts from any one of the landing positions, it will reach the shellcode.

[2]The size of shellcode ranges from dozens to hundreds [12].

Alignment size	Type A	Type B	Type C	Type D
64 kbytes	√	√	√	√
32 bytes	√	√	×	√
8 bytes	√	√	×	×
4 bytes	√	×	×	×

Table 1: Relationship between layout types and alignment size.

We summarize the relationship between four heap object structures and the memory alignment boundaries in table 1. When the alignment size is 64K bytes, all four heap object structures can be used. More generally, all four heap object structures can be used as long as the alignment size is larger than the size of the shellcode. When the alignment size is smaller than the size of the shellcode, the Type C layout does not work anymore, but the Type D is still effective.

In the new attack discussed in this paper, the sprayed heap objects are mostly filled with bytes that cannot be treated as NOP sled or bytes that cannot be interpreted as legal x86 instructions. NOZZLE can only find memory blocks that have a normalized surface area much lower than its threshold.

3.3 Surface area calculation

Our calculation involves the following variables: heap memory block size $size_{block}$, alignment size $size_{alignment}$, shellcode size $size_{sc}$, and normalized attack Surface Area NSA. We use NSA_{typeX} to represent the normalized attack surface area of Type X.

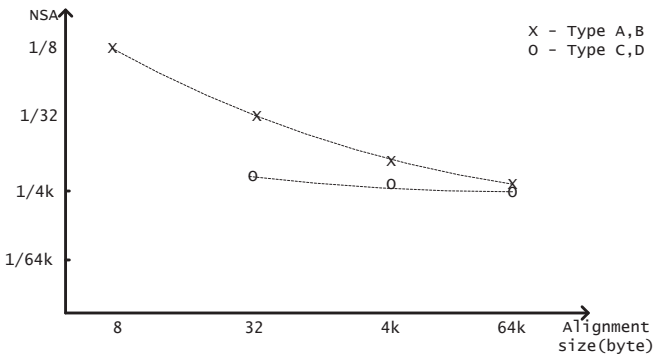


Figure 5: Normalized attack surface.

$$
\begin{aligned}
NSA_{typeA} &\approx NSA_{typeB} \\
&\approx \frac{\frac{size_{block}}{size_{alignment}} + size_{sc}}{size_{block}} \\
&= \frac{1}{size_{alignment}} + \frac{size_{sc}}{size_{block}} \\
NSA_{typeC} &\approx \frac{size_{sc}}{size_{alignment}} \\
NSA_{typeD} &\approx \frac{\frac{size_{sc}}{size_{alignment}} + size_{sc}}{size_{block}} \\
&= \frac{size_{sc}}{size_{block}} \times (1 + \frac{1}{size_{alignment}}) \\
&\approx \frac{size_{sc}}{size_{block}}
\end{aligned}
$$

From the formulas we can see that

- The NSA of Type A and B consists of two parts. The first term of the NSA is only relevant to alignment size, and the second term is relevant to both shellcode size and block size. The NSA of Type A and Type B increases when the alignment size or block size decrease, or when the shellcode size increases.
- The NSA of Type C is only relevant to the size of shellcode and memory block size. The NSA is proportional to the size of shellcode, and is inversely proportional to the size of the memory block size.
- The NSA of Type D is more complex, but it is clear that the NSA is inversely proportional to the size of the memory block. We also found that the NSA of Type D is much smaller than that of Type A and B.

There are three independent variables in these formulas and the function graph is hard to plot. To draw the graph, we must fix two of them. We assume that the heap memory block size is 1M bytes and the shellcode size is 256 bytes. Figure 5 shows the function graphs. The X-axis indicates the alignment size in bytes and the Y-axis indicates the normalized attack surface area (NSA). To simplify the calculation, we assume all the size of all instructions is one. Therefore, the surface area of practical samples may be two or three times of the theoretical value. As is showed in Figure 5, the normalized attack surface of all new heap objects is lower than the threshold of NOZZLE (50%).

3.4 Detecting Heap Taichi attacks

Enhanced NOZZLE *detection.*

From the above discussion, we can see that the assumptions made by NOZZLE are not necessary for a reliable heap-spraying attack.

NOZZLE can be enhanced to detect some of the new attacks by considering the effect of memory allocation granularity. The key is that all the landing positions should not be treated as the same. Instead, an enhanced NOZZLE algorithm should count the numbers of landing positions on each offset inside an "alignment-size segment" and record these numbers into an array. For example, on a 64K-byte aligned system, in a 1M-byte heap memory block, the three landing positions at `0x00c0c`, `0x10c0c`, `0x20c0c` number the count at `0x0c0c` as three. In the example of case study, the array at offset `0x0c0c` is counted as 8. Then we calculate the success rate on each offset. In the example of case study, the success rate on `0x0c0c` is $8 \div 8 = 100\%$ and on other positions the success rates are $0 \div 8 = 0\%$. Any success rate over 50% means a potential threat that may trigger a shellcode with a high success rate. The improved NOZZLE report a potential heap-spraying attack when it finds an offset with success rate over 50%.

However, the enhanced algorithm cannot deal with the Type C and D attack, where there are many copies of shellcode in one heap memory block. The landing positions are different from each other when analyzed statically because that they connect to different shellcode copies and these shellcode copies are not connected in the CFG. So, in Type C and D attack, the enhanced NOZZLE calculates the success rate at offset `0x0c0c` as $1 \div 8 = 12.5\%$. We report our evaluation results of this enhanced algorithm in Section 4.2.

Heap memory allocation in finer granularity.

The main problem behind this new type of attack is the predictability of heap addresses resulted from the coarse granularity of memory allocation. So a natural solution to prevent Heap Taichi attacks and similar attacks is to aligning memory allocation at a smaller-sized boundary. But we found it not easy to achieve in our experiments, because several application-level libraries align allocated memory objects by themselves.

There are many heap managers on different levels of a program, each of which has its own heap management strategy. For example, at the kernel level, there are "heap manager" in Windows, "SLUB allocator" in Linux, and Address Space Layout Permutation (ASLP) [26] in Linux. At the library level, there are libraries like jemalloc [23] and tcmalloc [35]. At the program level, we found that Firefox implemented a memory allocator based on object lifetimes named "JSArena" [25]. The heap manager on each level always manages its own "chunks" and also tries to get the chunks aligned on its own boundaries. Therefore, the granularity enforced by lower levels may be ignored in higher levels. For instance, jemalloc wraps `VirtualAlloc` and keeps its chunks aligned at 2M-byte boundaries. If the `VirtualAlloc` returns a memory block not aligned at the 2M-byte boundary, jemalloc frees the chunk and repeats the allocation until the returned memory block is aligned at the 2M-byte boundary.

To our understanding, the main reason for user-level alignment is performance. However, our performance evaluation (discussed in Section 4.3) showed that the gain in performance by the user-level alignment is not significant (less than 5% in our experiment). Therefore, memory managers at all levels should use finer memory allocation granularity for better security, a trade-off by sacrificing a limited amount of performance.

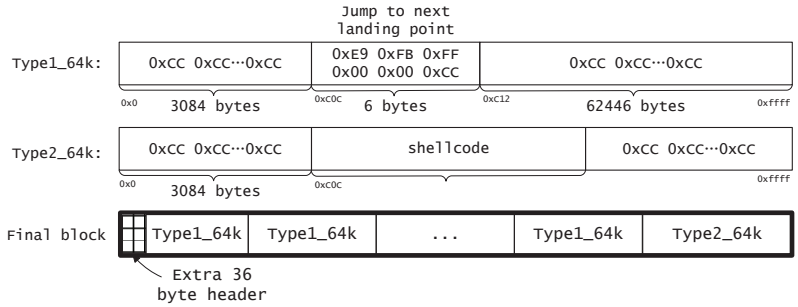


Figure 6: A sample structure of memory blocks with little surface area.

```
1  function heapspray(){
2   var scstring = unescape("%u9090...");
3   var alignment_size = 0x10000;
4   var pre_len = 0x00000c0c;
5   var post_len = alignment_size - 0x00000c0c - 0x6;
6   var head_offset = 0x24;
7   var jmp_str = unescape("%uFBE9%u00FF%uCC00");
8   var type1_str = CreateCCstringwithsize(pre_len) + jmp_str + CreateCCstringwithsize(post_len);
9   var type2_str = CreateCCstringwithsize(pre_len) + scstring +
       CreateCCstringwithsize(post_len + 0x6 - scstring.length * 2);
10  var type1_total_str = DuplicateStr(type1_str, 15);
11  type1_total_str = type1_total_str.substr(head_offset / 2,
                      type1_total_str.length - head_offset / 2);
                      // cut off the header bytes
12  var m = new Array();
13  for(i = 0; i < 200; i++)
14     m[i] = type1_total_str + type2_str;
15 }
```

Figure 7: Sample JavaScript spraying heap with Type A blocks.

4. EXPERIMENT AND EVALUATION

In this section, we describe our experiments of Heap Taichi, which generated heap objects that can bypass existing detection mechanisms. We also measure their normalized attack surface with different alignment sizes in the experiments.

4.1 Case study: A sample JavaScript code creating Type A heap objects

In this section, we give a JavaScript example of spraying a browser's heap with our Type A heap objects. This attack can also be done in other languages, including VBScript and ActionScript.

Figure 6 illustrates the structure used in this example. The malicious heap object's size is 1M bytes, consisting of two types of 64K-byte memory blocks. The first type only contains jump instructions at the landing positions, pointing to the landing position in the next block. The second type of block contains the shellcode at the landing position. We use the address `0x0c0c0c0c` as the jump target in step two of the attack. According to our analysis in Section 3, the landing position is at the offset `0x0c0c` of each 64K-byte block. We construct the final block by concatenating 15 type-1 blocks and one type-2 block, forming a heap object of 1M bytes. Note that each heap object allocated by Javascript has a 36-byte header (a Windows heap management header and a Javascript heap management header); we need to remove 36 bytes at the beginning of the final block, so that the offsets of landing positions will not be shifted by the header.

Figure 7 shows a piece of JavaScript code that implements a Heap Taichi attack, performing the heap object construction and heap spraying. The function `CreateCCstringwithsize` is used to create a string filled with value `0xCC` and the function `DuplicateStr` is used to create a long string. We fill the blocks with `0xCC`, because it is the opcode of x86's `INT 3` instruction, regarded as a terminator of a sequence of shellcode by existing approaches. We can fill these blocks with random bytes, because they are not used anyway. Because JavaScript strings use unicode encoding where each character takes 16 bits, we need to divide the length measured in bytes by two to get the correct length of unicode strings. Line 7 constructs the `type1_64k` block, and line 8 constructs the `type2_64k` block. Then line 9 and line 10 prepare the first half of the final block. Thirty six bytes are cut from the first half to accommodate the heap header. Finally, the heap is sprayed in line 12 to line 14 by an array of 200 strings containing the final block, taking up 200M bytes of the browser's heap region.

The `scstring` is filled with shellcode that libemu [9] cannot detect, which is captured by our drive-by download monitoring system [47]. The main reason that libemu cannot detect such shellcode is that libemu just emulates shellcode and once the shellcode includes instructions like `xor eax, [edi]` where register `eax` can only be determined at run-time, libemu cannot work well. For more evasion techniques, we refer readers to [29]. We extracted 44 shellcode pieces from those cached web pages, and 12 of them can't be detected by libemu. We choose a 236-byte shellcode to fill the `scstring`. Thus this script can bypass defending techniques based on libemu shellcode detection. We have also scanned this shellcode using 12 anti-virus products, and none of them could recognize it as a malicious code.

Sample ID	Heap Object Type	Alignment size
A64k	Type A	64k bytes
B64k	Type B	64k bytes
C64k	Type C	64k bytes
A4k	Type A	4k bytes
B4k	Type B	4k bytes
C4k	Type C	4k bytes
A32	Type A	32 bytes
B32	Type B	32 bytes
D32	Type D	32 bytes
A8	Type A	8 bytes
B8	Type B	8 bytes

Table 2: Samples used in surface area measurement

Block type	Alignment size			
	8 bytes	32 bytes	4K bytes	64K bytes
Type A	14%	3.6%	0.030%	0.0068%
Type B	25%	3.6%	0.030%	0.0068%
Type C			0.0055%	0.0054%
Type D		0.015%		

Table 3: Normalized attack surface area in our experiments

In our experiment, we modified a cached drive-by download web page by replacing its heap-spraying script with the one shown in Figure 7. Then we browsed the page using IE6 on Windows XP. The script reliably executed the shellcode, which downloaded and installed a bot on the victim machine.

4.2 Surface area measurement experiments

We build several example heap blocks of all the four heap structures and various alignment sizes, shown as Table 2. For example, A64k is the one given in the last subsection. B64k uses type B structure, a modified version of A64k with different jump instructions. C64k uses Type C structure, which can be achieved by replacing all `type1_str` with `type2_str` in our example JavaScript. The shellcode used in our experiment has 236 bytes, including 101 instructions. Its maximum attack surface area is 56.

To measure the normalized attack surface area (NSA), we implemented NOZZLE's surface area measurement algorithm. Table 3 summarizes the measured NSAs. We also plotted the results in Figure 8, where the Y-axis indicates the NSA of the attack vectors and the X-axis indicates the test cases. In Figure 8, we also marked several thresholds used in NOZZLE. When alignment size is 32 or higher, the normalized attack surface areas of the samples are far below the 50% threshold in NOZZLE.

When alignment size is 8, the Type C and Type D heap objects cannot be created. B8 exceeds the 20% "no false positive threshold" of NOZZLE, and A8 is on the border. The enhanced NOZZLE detection should cooperate with 8-byte or 4-byte heap allocation granularity. In the Type A and Type B objects, there are many landing positions connects to one copy of shellcode. The enhanced NOZZLE detects all of them and reports a heap spraying attack.

Also, we can see the difference in ratio between these results and the theoretical calculation in Section 3 is less than 3.0, which is close to the average instruction length. Therefore, the experiments confirm our theoretical analysis.

4.3 Performance of fine-grained memory allocation granularity

To evaluate the performance of 8-byte alignments, we built the Firefox 3.6.3 with jemalloc enabled and also modified jemalloc and SpiderMonkey with 8-byte randomization. Then we measured the modified Firefox's Javascript performance with Sunspider Javascript

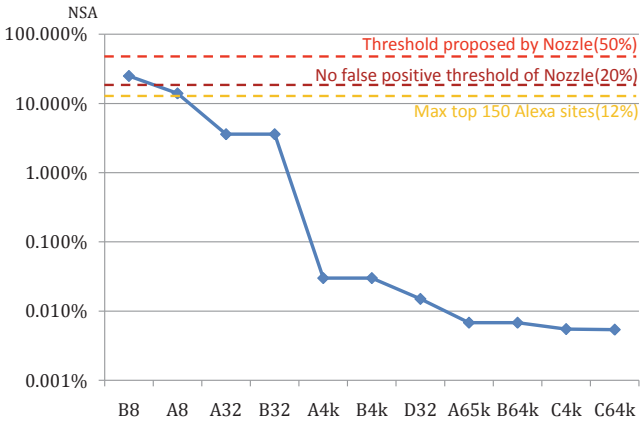


Figure 8: Sorted normalized attack surface area

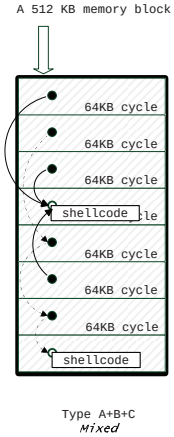


Figure 9: A sample mixed layout

Benchmark and V8 Benchmark. The result showed that the performance overhead is less than 5%. Researchers [26] also reported acceptable performance overhead of an 8-byte aligned randomization using other benchmarks.

5. DISCUSSION

5.1 Variations of Heap Taichi

Section 3 describes four basic memory layouts of Heap Taichi. Attackers may create new attacks by extending Heap Taichi.

At instruction level, attackers can replace those jump instruction with different instruction sequences, and fill arbitrary instructions between landing points and shellcode. At layout level, attackers can use the basic layouts to compose new layouts. For example, Figure 9 shows a "mixed" layout by combining Type A, B and C. Type C includes multiple copies of shellcode but keeps a small surface area; Type A and B layouts introduce more surface area but fewer shellcode copies. Attackers could use all these types in one heap block to balance these characteristics to evade detections.

5.2 Difficulty of detecting Heap Taichi attack

Under the 64K-byte alignment size, there are only 16 landing points in one 1M heap block as analyzed in Section 3. Attackers

could use mixed layouts similar to the example in Figure 9: place 3 to 5 shellcode pieces in one 1M heap block, and let landing points lead to any one of these shellcode pieces. In average, four landing points flow into one shellcode copy. Under this situation, there are no obvious anomalies in statistics compared to benign heap blocks, and it is very hard for methods like NOZZLE to detect this kind of attack without a high false positive rate.

As a consequence, to detect Heap Taichi under 64k memory alignment is as hard as to detect four shellcode copies in a 1M-byte heap object in real time. This could be a real challenge, and there is no practical solution which achieve both low false negative and low false positive so far.

6. RELATED WORK

6.1 Heap spraying with ASLR and DEP

The Address Space Layout Randomization(ASLR) technique [22, 33, 40, 44] is widely used in recent Windows versions such as Windows Vista and Windows 7. Analyses [40, 44] show that the randomization of heap area is quite weak. For each heap memory block, the system creates a five bit random number (between 0 and 31) and multiplies it with 64K, and then adds the product to the initial allocation base. This technique affects heap-spraying attacks, because it creates unpredictable gaps between the memory blocks. But attackers can deal with it by allocating a huge memory block and structure it carefully, so that the risk of landing in the gaps would be significantly reduced.

The ASLR-based defense is not effective on the new attack discussed in this paper. Because of the Windows memory allocation granularity, heap blocks are still aligned to 64K boundaries even after randomization. That means, the relative landing positions in each heap object is unchanged. As long as attackers can spray enough memory area using the heap region, the attack can still have a high success rate.

Data Execution Prevention (DEP) [1] is complementary to ASLR. It is an effective scheme to prevent an application or service from executing code from a non-executable memory region. Since shellcode is injected into non-executable memory region, most code injection attacks cannot work anymore when both DEP and ASLR are turned on. However, the attack techniques that can bypass DEP and ASLR are continually proposed. For example, Nenad Stojanovski et al. [41] showed that initial implementation of the software for DEP in Windows XP is actually not at all secure, and many attacks (such as return-to-libc like attack) can bypass DEP protection. Furthermore, Alexander Sotirov and Mark Dowd [39] implemented several exploitation techniques to bypass the protections and reliably execute remote code on Windows Vista. Dion Blazakis [15] illustrated two novel techniques (i.e., pointer inference and JIT spraying) to Windows Vista with IE8. Recently, during the PWN2OWN hacking contest 2010 [10], both IE 8 and Firefox 3 web browsers running on the Windows 7 system (both DEP and ASLR enabled) were successfully compromised. We believe that the attacks against DEP and ASLR cannot be completely avoided due to the vulnerabilities in operating systems or security-critical applications.

6.2 Heap-spraying attack and detection

Our approach is closely related to existing work on heap behavior manipulation, heap-spraying detection, as well as x86 executable code detection.

Heap behavior manipulation. A successful heap-spraying attack requires attackers to be able to predict the heap organization and, more importantly, locations of allocated heap objects. Sotirov [38] introduced a technique to use JavaScript to manipulate browser heap's layout, and implemented this technique into a JavaScript library for setting up heap state before triggering a vulnerability. Daniel et al. [19] developed a technique to reliably position a function pointer after a heap buffer that is vulnerable to buffer overflow. In this paper, we leverage a weakness on Windows heap allocation due to the large memory allocation granularity enforced on Windows systems, which makes heap allocation more predictable for attackers.

Executable code detection. Recent researches such as [28, 37] have proved that detecting arbitrary shellcode by static code features is difficult and even infeasible. In the context of network packets, several solutions [11, 30, 42] can detect executable code in the payload, but they cause high false positives in the context of heap objects [32], which makes them unsuitable for heap-spraying detection. In section 2.2, we have discussed several detection methods in detail.

6.3 Memory exploit detection and prevention

Note that heap spraying itself cannot directly cause the malicious payload to be executed. A successful attack needs another vulnerability to trigger the change of control flow to the sprayed heap. Detecting and preventing such vulnerabilities can stop heap spraying.

Buffer overflow is the common vulnerability exploited to redirect victim process's control flow. Traditional buffer overflow attacks target the pointer variables on stack or heap. A large number of solutions [45] have been proposed to address this problem. Among these efforts, address space layout randomization (ASLR) [2, 13, 14] provides general protection against memory exploits by randomizing the location of memory objects. It is now widely adopted in major operating systems. Note that address space layout randomization makes the location of memory objects, including heap objects, unpredictable, thus forcing heap-spraying attacks to inject a huge amount of heap objects containing code to increase the chance of success. This forms the basis for existing heap-spraying detection solutions.

Another common vulnerability exploited in browsers is integer overflow. Many integer overflow vulnerabilities are disclosed in recent years, and some integer overflow detection and prevention methods are proposed [16, 43]. Integer overflow leads to heap overflow in many cases, and heap spraying could construct step stones when exploiting these vulnerabilities.

In practice, it is very hard to eliminate all such vulnerabilities. Also, the runtime overhead prevents many of these approaches from being deployed widely. Therefore, the solution from this paper complements the approaches in memory exploit prevention.

7. CONCLUSION

Heap-spraying code injection attacks are commonly used in websites with exploits and drive-by downloads. This technique provides the attacker an easy-to-use code injection method which can be implemented in many type-safe languages. Since traditional heap spraying attacks require large number of NOP sled to increase the possibility of successful attacks, existing detection solutions mainly check for large amount of executable instructions on the heap.

By analyzing the operating systems' memory allocation mechanism, we found that the large amount of NOP sled is not necessary for heap spraying attacks if the memory alignment size is large enough. We introduced a new technique to launch heap-spraying

attack, which only injects a little amount of executable instruction, making it undetectable by existing approaches. We discussed the four basic types of attack modes and provide insight into the relationship between memory alignment size and heap spraying attack surface areas. We verified the technique by a proof-of-concept implementation. Even when the alignment size is 32 bytes, our attack can evade existing detection techniques. As a solution, we propose to enforce finer memory allocation granularity at memory managers of all levels, trading a limited amount performance for better security.

Acknowledgments The authors would like to thank the anonymous reviewers for their valuable comments. This work was supported in part by National Natural Science Foundation of China under the grant No. 61003216, National Development and Reform Commission under the project "A monitoring platform for web safe browsing", and Singapore Ministry of Education under the NUS grant R-252-000-367-133.

8. REFERENCES

[1] Microsoft Corporation. Data execution prevention. `http://technet.microsoft.com/enus/library/cc738483.aspx`.

[2] The PaX team. `http://pax.grsecurity.net`.

[3] Why is address space allocation granularity 64k? `http://blogs.msdn.com/oldnewthing/archive/2003/10/08/55239.aspx`.

[4] Microsoft Internet Explorer .ANI file “anjh” header BoF exploit, 2004. `http://skypher.com/wiki/index.php?title=www.edup.tudelft.nl/~bjwever/details_msie_ani.html.php`.

[5] Microsoft Internet Explorer DHTML object handling valuerabilities (MS05-20), 2004. `http://skypher.com/wiki/index.php?title=www.edup.tudelft.nl/~bjwever/advisory_msie_R6025.html.php`.

[6] Microsoft Internet Explorer IFRAME src&name parameter BoF remote compromise, 2004. `http://skypher.com/wiki/index.php?title=www.edup.tudelft.nl/~bjwever/advisory_iframe.html.php`.

[7] Microsoft Internet Explorer javaprxy.dll COM object vulnerability, 2005. `http://www.frsirt.com/english/advisories/2005/0935`.

[8] Microsoft Internet Explorer “msdds.dll” remote code execution, 2005. `http://www.frsirt.com/english/advisories/2005/1450`.

[9] libemu - shellcode detection, 2007. `http://libemu.carnivore.it`.

[10] Pwn2own 2010, 2010. `http://dvlabs.tippingpoint.com/blog/2010/02/15/pwn2own-2010`.

[11] P. Akritidis, E. P. Markatos, M. Polychronakis, and K. Anagnostakis. STRIDE: Polymorphic sled detection through instruction sequence analysis. In *Security and Privacy in the Age of Ubiquitous Computing*, 2005.

[12] C. Anley, J. Heasman, F. Lindner, and G. Richarte. *The Shellcoder’s Handbook: Discovering and Exploiting Security Holes*. Wiley, 2004.

[13] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceeding of 12th USENIX Security Symposium*, 2003.

[14] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of 14th USENIX Security Symposium*, 2005.

[15] D. Blazakis. Interpreter exploitation: Pointer inference and jit spraying. In *Blackhat, USA*, 2010.

[16] D. Brumley, T. Chiueh, R. Johnson, H. Lin, and D. Song. RICH: Automatically protecting against integer-based vulnerabilities. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS)*, 2007.

[17] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *POPL ’98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1998.

[18] CVE, 2007. `http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0038`.

[19] M. Daniel, J. Honoroff, and C. Miller. Engineering heap overflow exploits with JavaScript. In *Proceedings of the 2nd USENIX Workshop on Offensive Technologies*, 2008.

[20] T. Detristan, T. Ulenspiegel, and Yann_malcom. Polymorphic shellcode engine using spectrum analysis. *Phrack 11,57-15 (2001)*.

[21] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browser against drive-by downloads: Mitigating heap-srpaying code injection attacks. In *Proceedings of the 6th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2009.

[22] M. E.Russinovich and D. A.solomon. *Microsoft Wndows Internals, Fourth Edition: Microsoft Windows Server 2003, Windows Xp, and Windows 2000*. Microsoft Press, 2008.

[23] J. Evans. A scalable concurrent `malloc(3)` implementation for freebsd. In *BSDCan conference*, 2006.

[24] P. Fogla and W. Lee. Evading network anomaly detection systems: formal reasoning and practical techniques. In *CCS ’06: Proceedings of the 13th ACM conference on Computer and communications security*, 2006.

[25] D. R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Softw. Pract. Exper.*, 20(1):5–12, 1990.

[26] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *ACSAC’06: Proceedings of the 22th Annual Computer Security Applications Conference*, 2006.

[27] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS ’03: Proceedings of the 10th ACM conference on Computer and communications security*, 2003.

[28] J. Mason, S. Small, F. Monrose, and G. MacManus. English shellcode. In *CCS ’09: Proceedings of the 16th ACM conference on Computer and communications security*, 2009.

[29] M. Polychronakis, K. Anagnostakis, and E. Markatos. Emulation-based detection of non-self-contained polymorphic shellcode. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2007.

[30] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Network-level polymorphic shellcode detection using emulation. In *Proceedings of the 3rd Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2006.

[31] I. V. Popov, S. K. Debray, and G. R. Andrews. Binary obfuscation using signals. In *SS’07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, Berkeley, CA, USA, 2007.

[32] P. Ratanaworabhan, B. Livshits, and B. Zorn. NOZZLE: A defense against heap-spraying code injection attacks. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[33] J. Richter and C. Nasarre. *Windows via C/C++ 5th edition*. Microsoft Press, 2008.

[34] RIX. Writing ia32 alphanumeric shellcodes. *Phrack 11,57-15 (2001)*.

[35] P. M. Sanjay Ghemawat, 2005. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[36] SecurityFocus. Mozilla Firefox 3.5 ‘TraceMonkey’ component remote code execution vulnerability, 2009. `http://www.securityfocus.com/bid/35660`.

[37] Y. Song, M. E. Locasto, A. Stavrou, A. D. Keromytis, and S. J. Stolfo. On the infeasibility of modeling polymorphic shellcode. In *CCS ’07: Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007.

[38] A. Sotirov. Heap feng shui in JavaScript. In *Blackhat, USA*, 2007.

[39] A. Sotirov. Bypassing browser memory protections in windows vista. In *Blackhat, USA*, 2008.

[40] A. Sotirov and M. Dowd. Bypassing browser memory protections. In *BlackHat, USA*, 2008.

[41] N. Stojanovski, M. Gusev, D. Gligoroski, and Svein.J.Knapskog. Bypassing data execution prevention on microsoftwindows xp sp2. In *The Second International Conference on Availability, Reliability and Security (ARES)*, 2007.

[42] T. Toth and C. Kruegel. Accurate buffer overflow detection via abstract payload execution. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2002.

[43] T. Wang, T. Wei, Z. Lin, and W. Zou. IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, 2009.

[44] O. Whitehouse. An analysis of address space layout randomization on windows vista™. In *Symantec Advanced Threat Research*, 2007.

[45] Y. Younan, W. Joosen, and F. Piessens. Code injection in C and C++ : A survey of vulnerabilities and countermeasures. Technical Report CW386, Department of Computer Science, Katholieke Universiteit Leuven, 2004.

[46] A. Young and M. Yung. Cryptovirology: Extortion-based security threats and countermeasures. In *SP ’96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*, page 129, Washington, DC, USA, 1996. IEEE Computer Society.

[47] J. Zhuge, T. Holz, C. Song, J. Guo, X. Han, and W. Zou. Studying malicious websites and the underground economy on the chinese web. In *Proceedings of the 7th Workshop on the Economics of Information Security (WEIS’08)*, 2008.

SCOBA: Source Code Based Attestation on Custom Software *

Liang Gu‡, Yao Guo‡†, Anbang Ruan§, Qingni Shen§,Hong Mei‡
‡Key Laboratory of High Confidence Software Technologies (Ministry of Education),
‡Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing, China
§School of Software and Microelectronics, Peking University, Beijing, China
‡{guliang05,yaoguo,meih}@sei.pku.edu.cn,§{ruanab, shenqn}@infosec.pku.edu.cn

ABSTRACT

Most existing attestation schemes deal with binaries and typically require an exhaustive list of known-good measurements beforehand in order to perform verification. However, many programs nowadays are custom-built: the end user is allowed to tailor, compile and build the source code into various versions, or even build everything from scratch. As a result, it is very difficult, if not impossible, for existing schemes to attest the custom-built software with theoretically unlimited number of valid binaries available. This paper introduce SCOBA, a new Source COde Based Attestation framework, to specifically deal with the attestation on custom software. Instead of trying to obtain a know-good measurement list, SCOBA focuses on the source code and provides a trusted building process to attest the resulting binaries based on the source files and building configuration. SCOBA introduces a *trusted verifier* to certify the binary code of custom-build program according to its source code and building configuration. For custom-built software based on open-source distributions, we implemented a fully automatic trusted building system prototype for SCOBA based on GCC and TPM. As a case study, we also applied SCOBA to Gentoo and its Portage, which is a source code based package management system. Experimental results show that remote attestation, one of the key TCG features, can be made practically available to the free software community.

*This work is supported by the National Basic Research Program of China (973) under Grant No. 2009CB320703, the Science Fund for Creative Research Groups of China under Grant No. 60821003, National Key S & T Special Projects under Grant No. 2009ZX01039-001-001 and the National High-Tech Research and Development Plan of China under Grant No. 2007AA010304 and No.2009AA01Z139, and National Natural Science Foundation of China under Grant No. 60873238 and No. 60903178.
†corresponding author

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

Categories and Subject Descriptors

D.4.6 [**Operating System**]: Security and Protection—*Authentication, Invasive software*; D.2.4 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Security

Keywords

Remote attestation, custom software, trusted computing, free software, property-based attestation

1. INTRODUCTION

Many IT systems nowadays are conducted on open computer platforms across heterogeneous domains or over the public Internet. Entities in such an open, distributed and dynamic environment usually behave on their own behalf and may not trust each other for mission-critical operations or transactions. Remote attestation provides an important way to establish trust on parties in an open network. In Trusted Computing Group's trusted computing standard [23], remote attestation allows a challenging platform, usually referred to as a *challenger*, to verify the configuration integrity of a remote platform (i.e., an *attester*). Recent years have witnessed various evolutions out of the basic TCG attestation in many dimensions, including IMA [21], program semantics attestation [12], security policy enforcement [14], property attestation [4, 19], BIND [22], remote attestation on program execution [11], and so on.

Most of the existing remote attestation schemes are based on the integrity measurement of programs and configurations. The size of known-good measurements greatly limits the practicability of the existing attestation schemes. For example, free software and open-source software make it difficult, if not impossible, for existing attestation schemes to verify the genuineness of the corresponding binary code.

From the perspective of software deployment, there are usually two types of software: custom software and pre-packaged software. Many software nowadays are based on open-source software distributions, which greatly accelerate the software development process. However, since the users are in control of all the source files, they are able to tailor, configure and build their own executables to be deployed in their own environment. Even worse, they could also modify the source files at their own discretion, which would make

the situation worse for attestation schemes. Custom software can be configured and tailored according to the end user's requirements which cannot be predicted by the software provider. For example, Linux kernel can be configured and built for each and different platform, with different requirements specified by the users. The result is that even all users download a specific software from the same trusted source website, the executables they built themselves could all differ from each others.

Existing remote attestation schemes are not adequate to verify these custom-built software, mainly because it is impossible to hold a known-good measurement database for so many different programs of unpredictable versions. Although existing property-based attestation schemes [4, 19] introduced the concept of attesting programs based on their properties, these stated properties are still tied to the binary code. As a result, these property-based attestation scheme still need a giant known-good binary database, which is still not able to handle the custom software.

To deal with these challenges from custom software, we propose SCOBA, a new Source COde Based Attestation framework to solve the above problems and initiate an effort for applying remote attestation, one of the key TCG features, in the free software community. The rationale of our scheme is to link specified binary code of the customized software with its source code, and certify the generated binary code of the software according to both its source code and building configuration.

In order to validate the generated binary code of custom software, we introduce a Trusted Building System (TBS) to enable a trusted building process for compiling the custom software (Figure 1). In the trusted building process, the source code of the target program is tailored according to the end user's requirements and it is complied into binary code with the TBS, in which the binary code is bound with its corresponding source code and building configurations. The building process can be attested to prove the validation of the generated binary code. With the generated binary code and its corresponding source code as well as building configuration, a trusted verifier is introduced to certify the property of the custom software (step a and step b in Figure 1). At runtime, challenger may use the certificate to enable remote attestation on the custom software (step c and step d in Figure 1). So the trust chain of our attestation scheme can be built from the TPM to the building process, and finally to the attested custom software at runtime.

This paper makes the following key contributions:

- SCOBA solves the problem of known-good measurements database for custom software. With the proposed framework, it becomes practical to attest customized software in open networks. To our best knowledge, it is the first effort for employing attestation to enhance trust establishment on customized software, especially for customized open-source software.
- The proposed Trusted Building System enables another party to validate and certify the generated binary code of custom software according to its source code and building configuration. Existing solutions can not attest custom software, because it has no way of validating the binary code of custom-built software.
- The source code based approach is a more practical way to obtain the software property. As SCOBA binds the source code files and building configuration of the customized software, the trusted verifier may obtain its property by evaluating and testing these information.
- The trusted verifier in SCOBA serves as the verification agent, which can be customized to accommodate different types of software. As a result, SCOBA provides a flexible framework, which can be customized according to different types of software development process, as demonstrated by the case study on Gentoo.

The paper is organized as follows: we will give a brief introduction on background in Section 2. Section 3 introduces the design of SCOBA. Section 4 presents the implementation and evaluation on the prototype of SCOBA. Section 5 introduces the application of SCOBA in Gentoo. Section 6 introduces the related work. Section 7 discusses the possible solutions for improving our scheme and its application. Section 8 concludes the paper and discusses the future work.

2. BACKGROUND

2.1 Custom Software

From the perspective of software deployment, there are usually two types of software: custom software and pre-packaged software. Custom software, which is also called bespoke software, allows end user to design and implement software based on its own requirements. Pre-packaged software, or "off-the-shelf" software is released by software provider with specified configurations, such as the installation packages under Windows, rpm packages and Debian packages under Linux.

Sometimes custom software is referred to as configured software, or customized software, which is tailored or customized from the original version of delivered software. The custom software starts from an existing structure and it is flexible for various requirements. Free software and open software are the most widely available custom software. Like the Linux kernel, end users may modify and configure the free software at will to satisfy their specific requirements. However, such flexibility results large number of unpredictable versions for binary code of free software.

In this paper, we will consider two kinds of custom software: custom-built software–customizing without modifying the source code files; fully custom software–customizing with modifications on source code files. For the first type, the users customize the software distribution before building, but do not modify individual source code files that are attained from trusted parties. For the fully custom software, users can modify the source code of the original software. For the custom-built software, as it is supposed to have fixed source code, SCOBA is able to automatically certify this type of customized software. For the fully custom software, SCOBA may have to employ experts or more sophisticated certification techniques to certify these modified source code, such as model checking and testing.

When considering their tailoring platform, compilation platform and execution platform, the custom software deployment can either take place all on the same platform, or it could be performed on separated platforms. For example, the compilation and execution are carried out on separated platforms; the source code is tailored and built on a separated platform according to the end user's requirements.

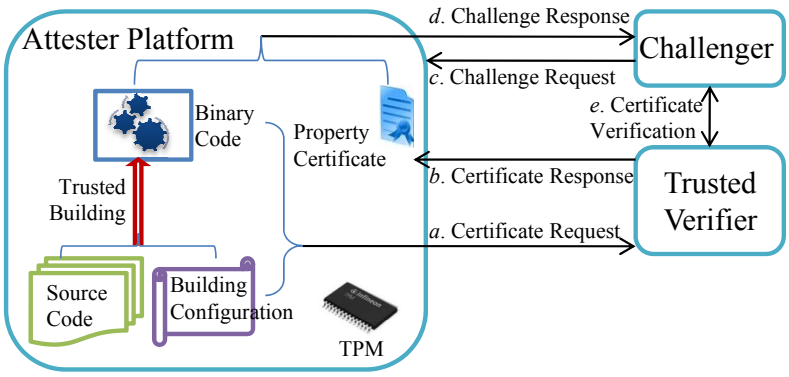


Figure 1: The overview of SCOBA

These approaches may require different designs and implementations of secure execution environment for the trusted building process in our scheme, and this will be discussed in Section 3.3 and Section 4. For most existing customized software, the first approach with the same platform is the most frequently used, thus we will focus on it most of the time and point out the difference if separated platforms might be used.

2.2 Dynamic Root of Trust

The TCG specifications introduce authenticated boot (or secure boot) to prove or guarantee that the system is booted into a secure state. The authenticated boot and secure boot provide a static root of trust based on the TPM. However, the static root of trust can not guarantee the security of a runtime system. With AMD's Secure Virtual Machine (SVM)[2] and Intel's Trusted Execution Technology (TXT)[13], it becomes practical to provide a dynamic root of trust for runtime system. The dynamic root of trust can strongly support a secure domain for dedicated system at runtime. Some studies for leveraging dynamic root of trust to provide secure execution environment have already been proposed, such as OSLO [15], Flicker [17] and TrustVisor [16]. For custom software delivered on the same platform, the end user may run the compilation process in the secure domain, which is supported by a dynamic root of trust.

3. SCOBA DESIGN

3.1 Attestation Framework

To provide remote attestation for custom software, we propose a new source code based attestation framework called SCOBA, which is illustrated in Figure 2. Instead of trying to obtain a list of known-good measurement list, SCOBA focuses on the source code, and provides a trusted building process to verify the resulting binaries according to the source files and building configuration.

Three parties are involved in SCOBA: the challenger, the attester and the trusted verifier. A typical scenario is as follows: the builder configures, tailors and builds a custom software P according to the challenger's requirement; the trusted verifier certificates the custom software by checking its source code, compiling configuration, binary code and records of building process; the builder delivers the custom software to the challenger with its certification; at runtime, the challenger wants to attest the property of this customized program; the challenger and attester will carry out the attestation procedure for this free software with the help of the trusted verifier.

Attester

The attester is the end user of a customized program P, which is executed on the attester platform. The attester customizes the source code of program P and takes a trusted building process to compile the source code into binary code. The trusted building process is introduced in Section 3.3. The attester employs a trusted verifier for property certification on the tailored program. The attester platform is supposed to be equipped with TPM.

Challenger

The challenger needs to attest the customized software being executed on the attester platform. The challenger requests the attester platform to return the integrity measurement and certificate of the target program. With these results, the challenger requests the trusted verifier to verify the certificate to determine the property of the target program.

Trusted verifier

The trusted verifier carries out two key tasks: property certification on customized software; runtime certificate verification. When the attester finishes the trusted building process, it requests a property certification on the customized program by sending all required source code, binary code and other records of trusted building process to the trusted verifier. The trusted verifier checks all these files and records to conclude with certain property for the customized program. At runtime, the challenger requests the trusted verifier to verify the certificate of the target program with specific integrity measurement. The trusted verifier can be a Trusted Third Party that issues property certificates and verifies certificates. The original provider of the program from which the customized software originates, can naturally serve as the role of trusted verifier.

3.2 Attester Platform

In the SCOBA framework, we assume that the attester platform (shown in Figure 2) is equipped with TPM, TXT[13] or SVM[2], the Secure Virtual Machine, the TCG software stack and an Attestation Agent, as well as a trusted building

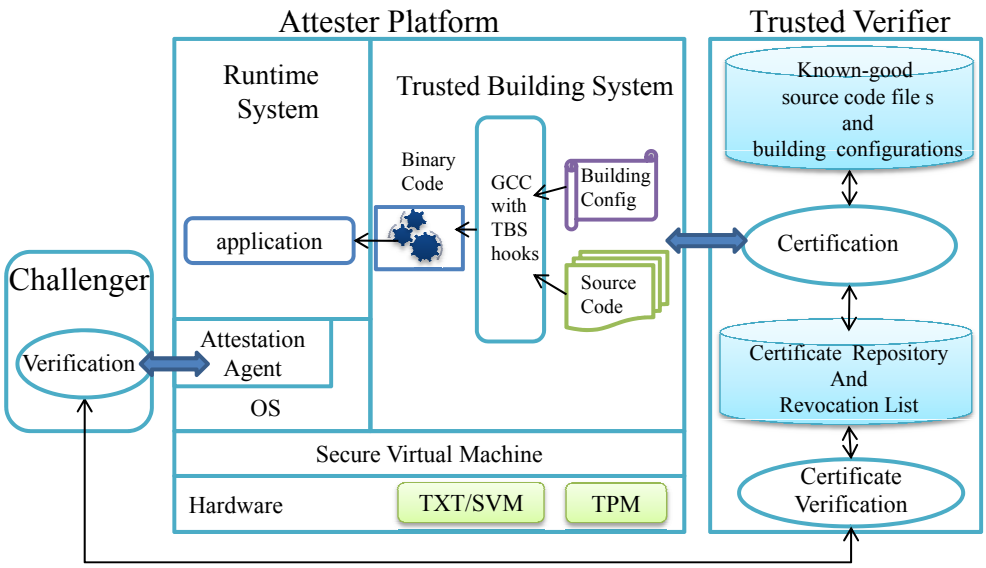


Figure 2: SCOBA Framework

system. The attester has privileges in controlling the software system on its platform. The attester platform may provide both static root of trust and dynamic root of trust. The challenger can establish trust on the integrity of a trusted domain based on Secure Virtual Machine with a dynamic root of trust.

Two separated domains are supported by the Secure Virtual Machine on the attester platform. One domain is a normal domain for ordinary operating systems. We introduce an attestation agent as the kernel module in the OS for runtime monitoring and recording target applications. The attestation agent is also responsible for communications between the challenger and attester platform. When the attestation agent receives the attestation request from the challenger, it records the states of target program and returns the target program's measurement and property certificate to the challenger. With the support of TPM and Secure Virtual Machine, the integrity of the attestation agent can be recorded for attestation each time before it starts.

The other domain is a secure domain that hosts a Trusted Building System (TBS). The TBS provides a trustworthy process for building these customized source code into binary code. The Secure Virtual Machine leverages the TXT/SVM facilities to provide a trusted domain for TBS. The building process can be attested to prove its trustworthiness.

The attester platform in Figure 2 is designed for customized software deployed on the same platform (Section 2.1). The TBS is supported by a dynamic root of trust. If it is carried out on separated platforms, the attester platform can have only the normal domain with attestation agent, while the TBS can be host on another separated platform. So TBS can run in a separated environment and its trust chain can be built on a static root of trust.

3.3 Trusted Building System

The Trusted Building System provides a trusted compilation process. A compilation process is considered trusted if the integrity of its execution can be attested to be without tampering. As a result, the compiled binary can be guaranteed to be generated from the input source code with specified configuration. In our scheme, the execution of TBS is protected by the secure domain, which can be set up at runtime based on a dynamic root of trust. TBS is supposed to be the minimal size for carrying out a compiling task and it is practical to implement the TBS with a thin OS and necessary compilation tools, e.g., the Linux From Scratch [1].

TBS records the states of all required proofs for program property certification. At the beginning of the trusted compiling process, TBS needs to record the building configuration. TBS records the state of compiled source code and output binary code files in a fine-grained and exact way according to their compiling order: the state of each source code file is recorded immediately before compilation; the state of each binary code file is recorded immediately when it is output by the compiler; meanwhile, TBS also binds the binary code file's measurement with the records of its corresponding source code files. In order to guarantee the integrity of these records, TPM is employed to record the states of all files.

3.4 SCOBA Procedure

The SCOBA procedure consists of three phases in our scheme: trusted building phase (trusted building in Figure 1), certification phase (step *a*, step *b* in Figure 1) and attestation phase (step *c*, step *d*, step *e* in Figure 1). TBS is responsible for the trusted building phase and records all required proofs for property certification on the target program. The trusted verifier issues the certificate according to these records generated by TBS. During the attestation phase, the challenger attests the target program with the help of the trusted verifier.

For a program P, its binary code files $F_e = \{f_{e_1}, f_{e_2}, ..., f_{e_i}\}$ is built from its corresponding source code files $S = \{f_{s_1}, f_{s_2}, ..., f_{s_j}\}$ with specified building configurations $C = \{c_{s_1}, c_{s_2}, ..., c_{s_j}\}$ and other dependent files $F_d = \{f_{d_1}, f_{d_2}, ..., f_{d_k}\}$,

where f_e is an executable file of P, f_{s_i} denotes a source code file for P and c_{s_j} stands for the building configuration of f_{s_j}. These building configurations may be stored in some script files $F_C = \{f_{c_1}, f_{c_2}, ..., f_{c_m}\}$, such as Makefile, .config files on Linux and the building command options. Other dependent files include mainly library files used during the building process.

3.4.1 Trusted Building Phase

In the trusted building phase, we bind the binary code of a program with its source code and building configuration. By leveraging a secure domain and TPM, a trust chain is built from the source code and building configuration to the generated binary code.

In order to construct the trust chain from TPM, two PCRs are employed in our scheme: one for authenticated boot of TBS (PCR_{ab}) and another for the trusted building process(PCR_{tbp}). These two PCRs are reset at the initialization stage of the secure domain. When the attester starts the trusted building process, a secure domain is initiated by the Secure Virtual Machine and the subcomponents of TBS are measured and recorded with an authenticated boot before it is about to run. After TBS finishes initialization, it starts to compile the target source code files and records the state of input and output files. TBS employs TPM to record the compilation process with *PCR_extend*. All inputs, intermediate outputs and generated codes are recorded to attest the compilation process.

As shown in Figure 3, a typical compilation task is carried out in roughly five stages: Preprocessing, Parsing, Translation, Assembling, and Linking. We may consider each stage as a transformation process with certain inputs and outputs. As shown in Figure 4, we may consider the compilation process as a sequence of transformations. The output of each prior stage can be the input of the next stage. The output of each stage may be in different forms according to different compiler implementations and building configurations. Usually the output includes specific data structure in the compiling process, and other supporting files. The TBS records the states of these output files and binds it with its corresponding inputs.

A transformation process T may have input files $F_{in} = \{f_{in_1}, f_{in_2}, ..., f_{in_i}\}$ from the prior transformation process, output files $F_{out} = \{f_{o_1}, f_{o_2}, ..., f_{o_j}\}$ and other dependent files $F_d(T) = \{f_{d_1}, f_{d_2}, ..., f_{d_k}\}$. For example, the object files generated by the Assembling process are the inputs files of Linking stage; the executable files produced at the Linking stage (Ⓕ) are output files; the library files at the Linking stage are dependent files.

As shown in Figure 4, four key points for monitoring and recording the trusted building process are identified:

- Point Ⓢ:the moment immediately before the initialization of the building process ;
- Point Ⓐ:the moment immediately before a transformation process T is going to run;
- Point Ⓑ:the moment immediately after a transformation process finishes.
- Point Ⓕ:the moment immediately after the trusted building process terminates ;

At each point, the monitoring and recording actions are required to execute according to the following rules:

- Point Ⓢ: For all script files in F_C that store the building configuration, TBS measures and extends them with TPM: $H_{c_k} = SHA1(f_{c_k})$, where $SHA1$ stands for an SHA-1 hash function; $H_C = SHA1(H_{c_1}||H_{c_2}||... ||H_{c_k})$; $PCR_{tbp\_extend}(H_C)$. For all source code files and corresponding configurations, TBS records their states and extends them into the TPM: $H_{s_i} = SHA1(f_{s_i} ||c_{s_i})$; $H_S = SHA1(\ H_{s_1}\ ||\ H_{s_2}\ ||...||\ H_{s_i}\ )$; $PCR_{tbp\_extend}(H_S)$. If these building configurations are stored in some configure files, these files are also recorded and extended by the TPM.
- Point Ⓐ: At the beginning of a transformation process T, all files in F_{in} are recorded and extended by TPM: $H_{in_i} = SHA1(f_{in_i})$; $H_{in}(T) = SHA1(H_{in_1}|| H_{in_2}||...||H_{in_i})$; $PCR_{tbp\_extend}(H_{in}(T))$. All dependent files, if exist, are also recorded and extended by TPM: $H_{d_i} = SHA1(f_{d_i})$; $H_d(T) = SHA1(H_{d_1}||H_{d_2}|| ...||H_{d_i})$; $PCR_{tbp\_extend}(H_d(T))$.
- Point Ⓑ: TBS records the state of all output files and employs TPM to extend their measurements: $H_{o_i} = SHA1(f_{o_j})$; $H_{out}(T) = SHA1(H_{o_1}||\ H_{o_2}||...||\ H_{o_i})$; $PCR_{tbp\_extend}(H_{out}(T))$. For an output file f_{o_j}, all input files which determine the generation of f_{o_j} are also recorded in set: $F_{in}(f_{o_j}) = \{f_{in_{j1}}, f_{in_{j2}}, ..., f_{in_{ji}}| f_{in_{ji}} \in F_{in}\}$.
- Point Ⓕ: At the termination stage of compilation, TBS records the states of all output executable files: $H_{e_i} = SHA1(f_{e_j})$; $H_e = SHA1(H_{e_1}||\ H_{e_2}||...||H_{e_i})$; $PCR_{tbp\_extend}(H_e)$. At last, TBS will employ the TPM to generate a signature on the final values in PCRs:

$$Quote_{tbp} = sig\{PCR_{tbp}\}_{AIK_{priv}}$$
$$Quote_{ab} = sig\{PCR_{ab}\}_{AIK_{priv}}$$

where AIK_{priv} is the private attestation key of TPM.

As the TPM extends all these records in sequence, an unbroken chain is established between the generated binary code and the source files with a given building configuration.

3.4.2 Certification Phase

After the trusted building process terminates, the attester sends a certificate request to a trusted verifier (step a) with the following messages:

$$\{F_e, S, F_C,\ H_e, H_S, H_C, H_{in}, H_d, H_{out}, PCR_{tbp}, PCR_{ab},$$
$$Quote_{tbp}, Quote_{ab}, AIK_{pub}, cert\{AIK_{pub}\}, SIG_M\}$$

where SML stands for Stored Measurement Log, AIK_{pub} stands for the public attestation key of TPM, $cert\{AIK_{pub}\}$ means the trusted certificate of TPM, $H_{in} = \{H_{in}(T_1), H_{in}(T_2), ..., H_{in}(T_i)\}$ is the set of input file records for all transformation processes, $H_d = \{H_d(T_1), H_d(T_2), ..., H_d(T_i)\}$ is the set of dependent file records for all transformation processes, SIG_M is the signature of these message which is generated with the session keys between the attester and trusted verifier. We assume that the communications between the attester and trusted verifier are protected. When the original provider of the customized software plays as the trusted verifier, it is not necessary to send all source code

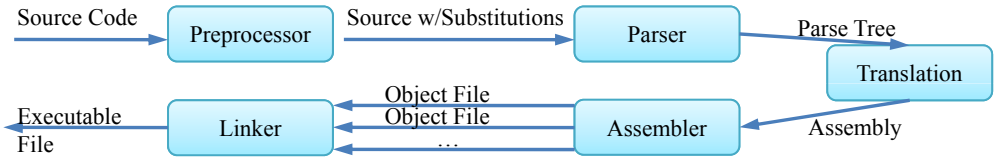


Figure 3: A typical compilation process in a trusted building phase.

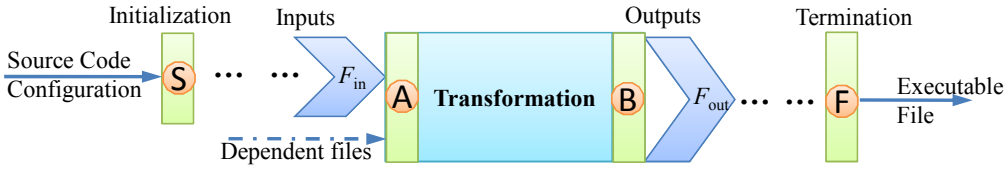


Figure 4: The detailed steps in a transformation process.

and configuration back to the trusted verifier. Only an update based on a standard version is required, such as the case with the Linux kernel patch.

The trusted verifier can conclude with certain properties for P by examining the received messages with following steps:

- First, the trusted verifier needs to attest the validation of TPM by checking its certification $cert\{AIK_{pub}\}$ and verify the integrity of messages.
- Second, it verifies the authenticated boot of TBS by checking PCR_{ab} and $Quote_{ab}$.
- Third, the verifier validates the trusted building process by checking $PCR_{tbp}, Quote_{tbp}$ according to F_e, S, H_C, H_e, H_S, H_{in}, H_d, H_{out}. Specially, the integrity of intermediate output files are required to be checked. For a input file f_{in_i}'s record H_{in_i} in transformation process (T_{i+1}), H_{in_i} should be consistent with its record H_{o_j} as output file in T_i.
- Finally, with all prior steps succeeded, the verifier will examine the source code, building configurations and binary code to determine the property of P. We will introduce a semantic approach of certifying the building configurations and source code in **Section 3.5**. If any of the above steps fails, the certification phase terminates with failure.

With a successful result, the trusted verifier issues the certificate on P and returns it to the attester platform (step b in Figure 1):

$$cert(TV, p, H_e) = (H_e, H_S, H_d, H_C, p, sig\{H_e||H_S||H_d||H_C||p\}_{SK_{TV}})$$

where (PK_{TV}, SK_{TV}) is the key pair of trusted verifier TV for signature, p is a property, $cert(TV, p, H_e)$ denotes the property certificate for P. With the certificate $cert(TV, p, H_e)$, the trust chain is linked from the source code to the properties of generated binary code via a trusted building process.

3.4.3 *Attestation Phase*

In the attestation phase, we assume the attestation target program is P. The challenger first sends a challenge request to the attester with a nonce (step c in Figure 1). Then the attester platform employs the attestation agent to collect the state and certificate of P. These records are sent back to the challenger as the challenge response (step d in Figure 1). Then the challenger may verify the certificate with the help of the trusted verifier and concludes with an attestation result on P(step e in Figure 1). During the verification stage in the attestation phases, challenger only has to submit the collected certificates to the trusted verifier to verify its validation, and with a successful verification result, the challenger can verify the runtime measurements according to these certificates.

3.5 Property Certifying via Semantic Verification on Building Configuration

During the certification phase, the trusted verifier needs to certify the program by examining the source code and building configuration to judge the property of the generated binary code. The property of the binary code is strictly linked with the building configuration. Let's take Gentoo Linux [9] as an example: Gentoo Linux employs Use Variable Descriptions (Global/Local Use Flags) to indicate which software features are included, and finally generates packages with different properties. Meanwhile, the source code of different versions hold different properties. The trusted verifier maintains a database for recording the properties according to their source code versions and building configurations. The trusted verifier will use this database to check the received source code and building configurations to determine the property of generated binary code.

It is straightforward to manually examine the building configuration according to standard configurations and limitations. However, it involves a lot of unnecessary effort, and it may involve human faults when it comes to a large number of configurations. With the cryptographic hash functions, it is also possible to automatically examine the hash values of the building configuration files, when only limited and predictable configurations exist according to involved program properties. However, the building configurations may have

huge amount of possible candidates and sometimes even infinite. When an option can be set as a floating value, the number of hash values for possible configuration files are almost infinite.

Fortunately, the building configurations are usually organized in a well-defined form, such as the Makefile, command options and .config files. Thus it is practical to examine the building configurations in a *semantic* way. We may consider a building configuration file as a collection of option pairs $< option, value >$. The trusted verifier has a set of criteria items $< option, criteria, operation, p >$ according to a specific program property p. The *operation* is determined by the type of option *value*. For example, the possible operations for integer values or floating values can be *equal*,*unequal*,*smallerthan*, etc. The operation set for all criteria can be determined according to the syntax of configuration file. In order to check whether the building configurations satisfy a certain property, an automatic process can be carried out to compare the option *value* with corresponding *criteria* according to specified *operation*. When the criteria of a specific program property is satisfied, the trusted verifier can conclude that the building configurations are with the property.

To perform semantic attestation on software configurations, we can apply a similar approach recently proposed in [24].

3.6 Property Certificate Revocation

It is possible that a program P, which is built based on a specific version of source code and building configuration, may be later found to be vulnerable or erroneous. So the trusted verifier needs to maintain a certificate revocation list to be able to revoke the corresponding property certificate. Once a program is identified as vulnerable, the corresponding certificate is added into the revocation list. During the attestation phase, the challenger is required to first check whether the property certificate is in the revocation list at certificate verification step (step e in Figure 1).

4. SCOBA IMPLEMENTATION FOR OPEN-SOURCE SOFTWARE

The proposed SCOBA framework could be applied to general custom software, however, it is most suitable for custom open-source software, where automatic attestation could be provided based on the open-source distributions. In this section, we apply SCOBA specifically to deal with custom open-source software, where users are allowed to tailor and configure the downloaded software, but are not allowed to modify the specific source code files. For cases of modifying source code, we will discuss it in Section 7.

We implement a prototype of this SCOBA framework to demonstrate its practical usage. Particularly, we focus on the customized open-source software on the Linux platform.

4.1 Attester Platform

We employ XEN [3] supported by the TXT facility as the Secure Virtual Machine. Ubuntu Linux is chosen as the operating system to host our prototype. In our implementation, we use Linux to run in two different domains of XEN: one is for ordinary applications in the ordinary domain of XEN and another is for the Trusted Building System in a protected domain. For the ordinary one, we introduce the attestation agent module as a Linux Security Module to monitor and record the execution of applications. For the TBS, we configure the Linux kernel via Linux From Scratch to get a minimal kernel to support the compilation tools, which carries out a trusted building process. We employ TXT to dynamically set up the secure domain for TBS [5].

4.2 Trusted Verifier

Trusted verifier maintains following repositories: a repository of known-good source code files and building configurations, a certificate repository, and a revocation list.

The known-good repository helps the trusted verifier to certify customized software. The known-good repository also records the properties of a software with specific source code files and building configurations for certain versions. The trusted verifier can automatically obtain the property of the target customized software. The certificate repository holds the records of all issued certificates and revocation list. The trusted verifier employs it to finish the certificate verification.

4.3 Trusted Building System

The Trusted Building System is the core of our scheme, and we will study its implementation based on GCC (GNU project C and C++ compiler) on Linux.

The GCC compilation process normally involves four steps: preprocessing, compiling, assembling and linking. The preprocessing step usually does not involve intermediate outputs, so TBS only has to monitor the intermediate outputs of following steps: compiling, assembling and linking. At the beginning of the above steps, we insert hooks into *gcc*, *as* and *ld* to monitor the inputs and outputs of these transformation processes. These hooks employ TPM to record the states of these inputs and outputs, and extend these records with the *PCR_extend* operation. At the end of the compilation process, TPM is invoked to generate quotes on these recorded proofs. In order to counter the "Time-of-measurement and Time-of-use" issue, we employ a similar mechanism as IMA [21] to deal with this problem.

4.4 Evaluation

We evaluate our prototype of TBS on a Lenovo ThinkPad X60 laptop with Intel Core 2 CPU T5500 @ 1.66GHz, and 1GB memory. We build a number of open-source applications with and without the proposed prototype, and the performance comparison is shown in Table 1.

In the table, we show the number for source code files, compilation time before and after applying the proposed scheme for each application. The cost for recording these proofs are roughly proportional to the number of source files in each application. The results show that TBS incurs roughly 2-4X slowdown on the evaluated benchmarks. The exception is TPM tools, which has an overhead of almost 15X because it involves only a handful of source files, thus the compilation time is relatively very short.

The overhead is pretty significant because of the large amount of TPM extend operations and low computation capability of TPM. However, the cost is still acceptable in practice, as TBS is only executed once for each build immediately before the certification.

Table 1: Comparison of compilation time before and after applying the proposed scheme.

applications	# of source code files	GCC-4.4.2	GCC-4.4.2 with TBS hooks
TPM-tools-1.3.4	59	14402 ms	209814 ms
Openssl-0.9.8k	1267	158106 ms	1318902 ms
Gmp-4.3.0	898	160279 ms	646499 ms
Trousers-0.3.1	326	118463 ms	345175 ms
Tboot-20090330	429	133646 ms	405173 ms
Linux-2.6.30	23214	7007143 ms	29034100 ms

5. CASE STUDY: APPLYING SCOBA TO GENTOO

Gentoo [9] is a free operating system based on either Linux or FreeBSD that can be automatically optimized and customized for just about any application. Most applications are distributed in the form of source code in Gentoo and its package management tool Portage is responsible for building and installing these applications. We can apply SCOBA straightforwardly to Portage to support attestation on custom software in Gentoo systems. Besides our modified GCC compilation tools with TBS hooks, we may also leverage Portage to provide a more flexible monitoring and recording mechanism for attesting customized software in Gentoo.

Portage is the heart of Gentoo, and performs many key functions. It serves as the software distribution system for Gentoo. It can maintain a local Portage tree which contains a complete collection of scripts that can be used by Portage to create and install the latest Gentoo packages. Portage is also a package building and installation system. It will build a custom version of the package to the user's exact specifications, optimizing it for the hardware and ensuring that only the optional features in the package that the users want are enabled.

Portage is characterized by its main function: compiling from the source code of these packages that the user installs. In doing so it allows customizing package functionalities to the user's own wishes, and customizing all packages to the systems specifications. In order to accomplish this, several functionalities are provided. Functionalities concerning managing the system are: allowing parallel package version installation, influencing cross-package functionalities, managing an installed-packages database, providing a local ebuild (explained later) repository, and syncing of the local Portage tree with remote repositories. Functionalities concerning installing the individual package are: specifying compilation settings for the target machine, and influencing specified package components.

The basis for the entire Portage system is the *ebuild* scripts. They contain all the information required to download, unpack, compile and install a set of sources, as well as how to perform any optional pre/post install/removal or configuration steps. An ebuild is a specialized bash script format created by the Gentoo Linux project for use in its Portage software management system, which automates compilation and installation procedures for software packages. Each version of an application or package in the Portage repository has a specific ebuild script written for it. The script is used by the *emerge* tool, also created by the Gentoo Linux project, to calculate any dependencies of the desired software installation, download the required files (and patch them, if necessary), configure the package, compile, and perform a sandboxed installation. Upon successful completion of these steps, the installed files are merged into the live system, outside the sandbox.

Base on the characteristics of Gentoo, we can easily extend TBS into the Gentoo Portage, and hence support trusted building in Gentoo. There are a number of different functions that we can define in ebuild files that control the building and installation process of the package. Hence, we can add specific TBS hooks in the call-sites of these functions in Portage to perform monitoring on the trusted building and installing procedure. These functions include:

- *Pkg_setup*: This function can perform any miscellaneous prerequisite tasks. This might include checking for an existing configuration file. We can add functions to initialize a trusted and isolated environment for the building procedure.
- *Src_unpack*: This function unpacks the sources, applies patches, and runs auxiliary programs such as the autotools. We can initialize the trusted measurement repository for all the source codes. Normally, the source codes are distributed in a single compressed package (e.g. tar file). Hence we should first generate the genuine measurement value for each file in the package (e.g. source codes, configuration files, etc.) from the signed measurement value of the source code package.
- *Src_compile*: This function configures and builds the package. We can integrate our trusted building mechanisms here.

Moreover, the following functions can be modified for implementing advanced trusted installation procedures, e.g. generating proof chains or related certificates.

- *pkg_preinst*: The commands in this function are run just prior to merging a package image into the file system.
- *Src_install*: This function installs the package to the destination.
- *Pkg_config*: This function sets up an initial configuration for the package after it's installed.
- *Pkg_postinst*: The commands in this function are executed immediately following merging a package image into the file system.

The package repository of Gentoo is in the best position to serve as the trusted verifier. Besides the package data, the package repository also maintains the corresponding property information in order to certify customized software. In

order to support runtime certificate verification, the package repository maintains the certificate repository and revocation list.

6. RELATED WORK

Since TCG attestation was introduced as a key feature in the TCG specification[23], many remote attestation schemes have been proposed in the literature. Terra [7] employs a Trusted Virtual Machine Monitor (TVMM) to transform a tamper resistant hardware platform into multiple isolated virtual machines (VMs). With the protection of the trusted hardware, TVMM offers both the open-box VM and the closed-box VM. The attestation in TVMM only measures the programs before their executions and is not able to check their behaviors after attestation. As an extension of TCG attestation, IMA [21] employs a loading time integrity measurement mechanism which measures all software components including BIOS, the OS loader, the operating system, and programs at the application layer. The limitation of integrity-based attestation such as IMA is that it checks at the loading time. Since there exists a gap between time of measurement and time of execution, loading time integrity does not necessarily lead to stronger security assurance. As a follow-up of IMA, [20] employs IMA to enforce remote access control by attestation.

Property-based attestation [4, 19, 18] was introduced to provide a scalable attestation framework to support privacy preserving for the attester platform. A trusted third party is introduced to exam the runtime measurements and judge the property of the target platform. The challenger only verifies the property certificate to conclude the attestation result and the configuration information of the attested platform is preserved. Existing schemes of TCG attestation and property-based attestation are based on the known-good measurements of these attested programs.

Haldar et al. [12] introduced a semantic attestation mechanism based on the Trusted Virtual Machine (TVM). The TVM based semantic attestation mechanism enables the remote attestation of high-level program properties. Shi et al. proposed a fine-grained attestation scheme called BIND [22]. It provides evaluation interfaces to attest the security-concerned segments of code. Jaeger et al. [14] introduced the Policy-Reduced Integrity Measurement Architecture (PRIMA) based on the information flow integrity checking against the Mandatory Access Control (MAC) policies. Program execution attestation introduced in [11] is to attest whether a program is executed as expected. These semantic attestation mechanisms still require a know-good binary code repository.

However, most of the existing schemes are still based on binary attestation, as it plays an important role for authentication on software. As the binary attestation involves verification on the measurement of binary code, most of existing schemes have to face the problem of keeping a huge known-good measurements database in practical solutions.

Trusted Execution Technology (TXT) and Secure Virtual Machine (SVM) are introduced to provide a trusted execution environment. Recent years, there are already several practices [8, 15, 17] exploiting TXT or SVM. Open Secure LOader (OSLO) [15] leverages the dynamic root of trust to implement a bootloader based on AMD $SKINIT$ instruction. Flicker [17] was introduced as an infrastructure for executing security sensitive code in complete isolation. It leverages the Secure Virtual Machine (SVM) of AMD processors and provides fine-grained attestation on program execution. LaLa [8] combines the latest hardware virtualization and trust technologies to deliver a more robust platform to support both instant-on system and a full-featured OS, and the flexible architecture enables a platform user to benefit from the advantages of a fast booting platform and a full-featured mainstream OS at the same time.

7. DISCUSSION

The proposed SCOBA framework could be applied to general custom software provided that a trusted verifier could be provided for all source files and configurations, which is not always practical. Here we discuss some of the limitations and possible enhancements of the proposed approach.

Selection of Trusted Verifier

It is important to choose the right party to play as the role of trusted verifier. In order to certify a customized software, the trusted verifier is supposed to have enough knowledge for carrying out the certification process. The provider of the original software holds the best position to serve as the role of trusted verifier for certifying the property of the customized software. However, when the original provider is not trusted or not available, a trusted third party can be employed and it should maintain a repository to store the property information of all known-good source code, which may come from different software providers, another trusted third party or trusted agent for software certification.

Automatic source code certification on custom-built software

For a custom-built program with only variant building configurations, the trusted verifier can employ semantic verification to automatically examine the building configurations. If the custom-built software does not make any modifications on the source code, the trusted verifier can maintain a repository of known-good source code files according to specific properties. In the certification phase, the proofs of trusted building process for the target custom software can be automatically analyzed to conclude its property.

Attestation on fully custom software

For fully custom software, users may modify the source code of the target custom software or even add new source code files into the software. It is difficult for a trusted verifier to automatically certify the modified source code. A straightforward way is to have experts manually checking these modifications and determine the property of the custom software. For programs with source code modifications at lower granularity (such as instructions), besides the manual verification on these modified codes, the trusted verifier can also employ more sophisticated certification techniques for automatic program certification, such as testing [6] and model checking. The certification on a whole customized software can be accomplished by certifying its software components [10]. The custom software may be built from scratch, and its source code files or subcomponents may come from other open source software. So it is possible to automatically certify these subcomponents from known software distributions.

Supporting semantic based attestation on custom software

The proposed scheme can serve as a building block for other types of semantic based attestation [12] on customized software. Different types of semantic attestation solutions may concern different properties of software. However, the integrity of a program is the basis for all different solutions. Our scheme provides the possibility to attest the customized software with unpredictable versions and configurations.

8. CONCLUDING REMARKS

In this paper, we introduce SCOBA, a source code based attestation scheme for custom software. SCOBA enables property attestation on custom software with unpredictable versions and building configurations. With a trusted building process, SCOBA binds the binary code of a program with its source code and building configuration. Then a trusted verifier is able to certify the generated binary code with the proofs from the Trusted Building System and determine the property of the target custom software by checking the source code and building configurations. Thus SCOBA links the trust chain between TPM to the runtime attested custom software. We implement a prototype of SCOBA based on GCC compilation tools and TPM. Experiments show that the performance is acceptable in practice. We also studies the application of SCOBA on Gentoo to support attestation on free software distributed in the source code form. With the support of SCOBA, it is possible for the free software community to employ remote attestation, one of the key TCG feature, to support trust establishment on applications in an open networking environment.

9. REFERENCES

[1] Linux From Scratch. `http://www.linuxfromscratch.org/index.html`.

[2] AMD. AMD64 Virtualization Codenamed "Pacifica" Technology–Secure Virtual Machine Architecture Reference Manual. Technical Report Publication Number 33047, Revision 3.01, AMD, May 2005.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, volume 37, 5 of *Operating Systems Review*, pages 164–177, Oct. 19–22 2003.

[4] L. Chen, R. Landfermann, H. Löhr, M. Rohe, A.-R. Sadeghi, and C. Stüble. A protocol for property-based attestation. In *STC '06*, pages 7–16, New York, NY, USA, 2006. ACM Press.

[5] J. Cihula. Trusted Boot: Verifying the Xen Launch. `http://www.linuxfromscratch.org/index.html`. Xen Summit 07 Fall.

[6] G. Fink and M. Bishop. Property-based testing: a new approach to testing for assurance. In *ACM SIGSOFT Software Engineering Notes*, volume 22(4), 1997.

[7] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra a virtual machine-based platform for trusted computing. In *SOSP 2003*, Bolton Landing, New York, USA, October, 2003.

[8] C. Gebhardt and C. Dalton. Lala: a late launch application. In *STC '09: Proceedings of the 2009 ACM workshop on Scalable trusted computing*, pages 1–8, New York, NY, USA, 2009. ACM.

[9] Gentoo. Gentoo Linux. http://www.gentoo.org/, 2009.

[10] A. K. Ghosh and G. McGraw. An approach for certifying security in software components. In *Proc. 21st NIST-NCSC National Information Systems Security Conference*, pages 42–48, 1998.

[11] L. Gu, X. Ding, R. H. Deng, B. Xie, and H. Mei. Remote attestation on program execution. In S. Xu, C. Nita-Rotaru, and J.-P. Seifert, editors, *STC*, pages 11–20. ACM, 2008.

[12] V. Haldar, D. Chandra, and M. Franz. Semantic remote attestation—a virtual machine directed approach to trusted computing. In *the Third virtual Machine Research and Technology Symposium (VM '04). USENIX.*, 2004.

[13] Intel Corporation. Intel trusted execution technology — preliminary architecture specification. Technical Report Document Number: 31516803, Intel Corporation, 2006. `ftp://download.intel.com/technology/security/downloads/31516803.pdf`.

[14] T. Jaeger, R. Sailer, and U. Shankar. PRIMA: policy-reduced integrity measurement architecture. In *SACMAT '06*, pages 19–28, 2006.

[15] B. Kauer. OSLO: Improving the security of Trusted Computing. In *Proceedings of the 16th USENIX Security Symposium*, 2008.

[16] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010.

[17] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: an execution infrastructure for tcb minimization. In J. S. Sventek and S. Hand, editors, *EuroSys*, pages 315–328. ACM, 2008.

[18] J. Poritz, M. Schunter, E. Van Herreweghen, and M. Waidner. Property attestation—scalable and privacy-friendly security assessment of peer computers. Technical Report RZ 3548, IBM Research, May 2004.

[19] A.-R. Sadeghi and C. Stüble. Property-based attestation for computing platforms: caring about properties, not mechanisms. *New security paradigms*, 2004.

[20] R. Sailer, T. Jaeger, X. Zhang, and L. v. Doorn. Attestation-based policy enforcement for remote access. In *CCS 04*, October 25-29, 2004.

[21] R. Sailer, X. Zhang, T. Jaeger, and L. v. Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, USA, August, 2004.

[22] E. Shi, A. Perrig, and L. V. Doorn. Bind: A fine-grained attestation service for secure distributed systems. In *2005 IEEE Symposium on Security and Privacy*, 2005.

[23] Trusted Computing Group. TPM main specification. Main Specification Version 1.2 rev. 85, Trusted Computing Group, Feb. 2005.

[24] H. Wang, Y. Guo, and X. Chen. Saconf: Semantic attestation of software configurations. In *ATC '09: Proceedings of the 6th International Conference on Autonomic and Trusted Computing*, pages 120–133, 2009.

Paranoid Android: Versatile Protection For Smartphones

Georgios Portokalidis*
Network Security Lab
Dept. of Computer Science
Columbia University, NY, USA
porto@cs.columbia.edu

Philip Homburg
Dept. of Computer Science
Vrije Universiteit
Amsterdam, The Netherlands
philip@cs.vu.nl

Kostas Anagnostakis
Niometris R&D
Singapore
kostas@niometrics.com

Herbert Bos
Dept. of Computer Science
Vrije Universiteit
Amsterdam, The Netherlands
herbertb@cs.vu.nl

ABSTRACT

Smartphone usage has been continuously increasing in recent years. Moreover, smartphones are often used for privacy-sensitive tasks, becoming highly valuable targets for attackers. They are also quite different from PCs, so that PC-oriented solutions are not always applicable, or do not offer comprehensive security. We propose an alternative solution, where security checks are applied on remote *security servers* that host exact *replicas* of the phones in virtual environments. The servers are not subject to the same constraints, allowing us to apply *multiple* detection techniques *simultaneously*. We implemented a prototype of this security model for Android phones, and show that it is both practical and scalable: we generate no more than 2KiB/s and 64B/s of trace data for high-loads and idle operation respectively, and are able to support more than a hundred replicas running on a single server.

Categories and Subject Descriptors

D.2.0 [**General**]: Protection mechanisms

General Terms

Design, Security, Reliability

Keywords

Decoupled security; Smartphones; Android

1. INTRODUCTION

Smartphones have come to resemble general-purpose computers: in addition to traditional telephony stacks, calendars, games and address books, we use them for browsing, reading email, watching videos, and many other activities that we used to perform on PCs. As software complexity increases, so does the number of bugs and exploitable vulnerabilities [17, 32, 20, 31]. Vulnerabilities in the past have allowed attackers to exploit bugs in the Bluetooth network stack to take over various mobile phones. More recently, Apple's iPhone and Google's Android platform have also shown to be susceptible to remote exploits [28, 24, 25] .

Moreover, as phones are used more and more for privacy sensitive and commercial transactions, there is a growing incentive for attackers to target them. For instance, smartphones can be used to perform online purchases, control bank accounts, store passwords and other sensitive information like social security numbers, *etc.* Phone-based payment for physical goods, services, mass transit, and parking is also provided by various companies like Upaid Systems, Black Lab Mobile, and others. Compromised smartphones can also be used to spy upon users, as they include a GPS sensor and a microphone that can be used to obtain a user's location or eavesdrop.

Smartphones will most likely become targets in the future, and while average users may not be willing – for the time being – to pay the cost (both in financial and performance terms) of securing their devices, *this is not the case for senior officials in industry, government, law enforcement, banks, health care, and the military*[1]. Smartphones are already an integral tool in many such organisations, but due to security and privacy concerns, and due to the lack of security mechanisms, administrators often resolve in limiting the functionality of employees' devices (like disabling WiFi connectivity and reception of SMS messages). *In this paper, we address the problem of security for smartphones for organisations and individuals that care deeply about the detection of attacks.* Our goal is to provide versatile security for smartphones, offering detection of a wide range of attacks including zero-day ones.

Deploying security mechanisms on already severely resource-constrained smartphones can be problematic. For instance, running a simple file scanner like ClamAV on the Android

*This work was done while the author was in Vrije Universiteit Amsterdam.

[1]A famous case in point was president Obama's 2008 struggle to keep his Blackberry phone after being told this was not possible due to security concerns. Eventually, he was allowed to keep an extra-secure smartphone.

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

HTC G1's data and application folders took approximately 30 minutes, and reduced battery capacity by 2%. Other work [6] has also shown that running a naive file scanning application on an HTC G1 is 11.8x slower than running it on single-core virtual machine (VM) running on a desktop PC. *We argue for a different security model that completely devolves attack detection from the phone.*

At a high level, we envision that security (in terms of attack detection) will be just another service hosted in the cloud, much like storage and email. Whether this is practical, or even feasible at the granularity needed for thwarting today's attacks has been an open research question, which we attempt to answer in this paper. More specifically, we propose running a synchronised replica of the phone on a *security server* in the cloud. As the server does not have the tight resource constraints of a phone, we can perform security checks that would be too expensive to run on the phone itself. To achieve this, we record a minimal trace of the phone's execution (enough to permit replaying and no more) which we then transmit to the server. The implementation of our security model is known as *Paranoid Android* (*PA*).

Our approach is consistent with the current trend to host activities in the cloud, including security-related functions. Oberheide *et al.* have explored AV file scanning in the cloud with [29] and [30], but file scans are not able to detect zero-days, remote exploits, or memory-resident attacks (all of which have targeted mobile phones in the past [20, 14, 31, 25]). One could argue that smartphone components are frequently coded in languages like Java that do no suffer from such attacks. But the runtime environments (JREs) used on smartphones are usually smaller, optimized versions of the original JRE (*e.g.,* Android uses the DEX Dalvik VM), which do not necessarily provide the same security and isolation guarantees , and can be themselves vulnerable to attacks. Furthermore, most platforms (including Android) offer native APIs for high performance applications that are vulnerable to a wider range of attacks.

Our solution builds on work on VM recording and replaying [11, 42, 26, 5, 12, 19, 37, 38, 23]. Previous work on PC systems, makes use of tailored VMs, and assumes ample and cheap communication bandwidth. Rather than recording and replaying at the VM level, we record the trace of a set of processes (running everything in a VM on the phone is not realistic on any current phone). In addition, we tailor the solution to smartphones, and compress and transmit the trace in a way that minimises computational and battery overhead. We also ensure that an attacker compromising a device cannot bypass the security measures applied at the server, and elude detection.

The main contributions of this paper are:

- A scalable smartphone security architecture that is able to apply multiple security checks simultaneously without overburdening the device.
- A prototype implementation of an execution recording and replaying framework for Android.
- Transparent backup of all user data in the cloud.
- A replication mechanism that guarantees the detection of an attack.
- Application transparent recording and replaying.

The remainder of the paper is organised as follows. The architecture of *PA* is discussed in Section 2, while implementation details of our prototype are given in Section 3. We evaluate the system in Section 4, and review related work in Section 5. Conclusions are in Section 6.

2. PARANOID ANDROID ARCHITECTURE

A high-level overview of *PA*'s architecture is illustrated in Figure 1. On the phone, a *tracer* records all information needed to accurately replay its execution. The recorded *execution trace* is transmitted to the cloud over an encrypted channel, where a replica of the phone is running on an emulator. On the cloud, a *replayer* receives the trace and faithfully replays the execution within the emulator. We can apply security checks externally, as well as from within the emulator, as long as they do not interfere with the replayed applications (*i.e.,* they do not perform IPC with replayed processes, modify user files, *etc.*). Provided we observe this rule of non-interference, we may even run additional processes or instrument the kernel. Furthermore, we use a network proxy to connect to the Internet, which allows us to intercept and temporarily store inbound traffic. The *replayer* can access the proxy to retrieve the data needed for replaying. This way the *tracer* does not have to retransmit the data received over the network to the replica.

2.1 Recording And Replaying

Recording and replaying a set of processes and entire systems has been broadly investigated by previous work [11, 42, 26, 5, 12, 19, 37, 38, 23, 16]. We will only briefly discuss how execution replaying is performed, while implementation specifics and various optimisations are discussed in Section 3.1. Readers interested in recording and replaying in general are referred to the above cited papers, and our technical report on *PA* [34].

A computer program is by nature deterministic, but it receives nondeterministic inputs and events that influence its execution flow. To replay a program, we need to record all these nondeterministic inputs and events. Such inputs usually come from the underlying hardware (*e.g.,* time comes from the HW clock, network data from the WiFi adaptor, location data from the GPS sensor, *etc.*), which a process receives mostly through system calls to the kernel. Thus, to replay execution the *tracer* records all data transferred from kernel to user space through system calls. The *replayer* then uses the recorded values when replaying the system calls on the replica. Note that we only replay process and not kernel execution. While this implies that *PA* may not be able to detect an attack against the kernel, most kernel vulnerabilities are only exploitable locally, which would require that the attacker first compromises a user process.

Beside system calls, operating systems (OSs) can also alter a process' control flow by using synchronous and asynchronous notification mechanisms such as signals. For instance, a signal may be sent to a process when a certain event occurs (*e.g.,* a timer expires). Signals that notify of serious errors (*e.g.,* a segmentation fault, or a floating point exception) are delivered synchronously, when the instruction that caused the error is executed. Consequently, they will be also generated by the OS on the replica. On the other hand, asynchronous signals can be delivered arbitrarily, and in fact most OSs (except real-time ones) do not even guarantee their delivery. To ensure that such signals are delivered

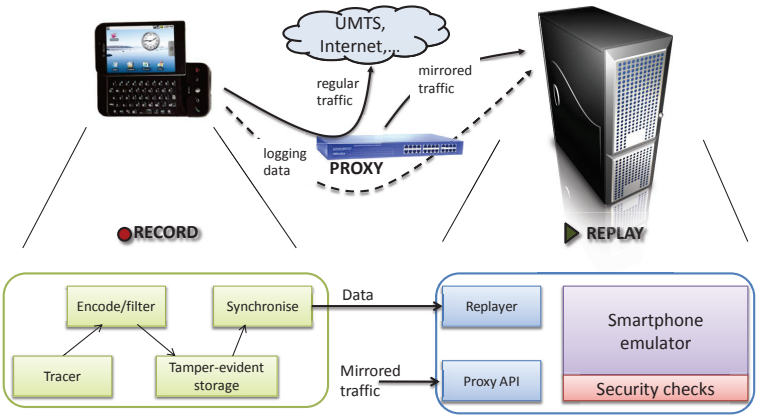


Figure 1: *Paranoid Android* **architecture overview**

at exactly the same time during replay, we defer their delivery until the target process performs a system call.

Concurrency and inter-process communication (IPC) can also be a source of nondeterminism. Two processes can exchange data using various mechanisms such as pipes, message queues, files, sockets, shared memory, and memory mapped files. Most of these mechanisms are implemented using system calls to send and receive data, therefore we implicitly support them by accurately replaying system calls.

This is not the case for shared memory and memory mapped files, since they can be accessed directly. When two or more processes use such objects to exchange data, they may affect one another in unpredictable ways, producing non-reproducible behaviour. In the case of threads, almost all process memory is shared. In the presence of shared objects, accesses on these objects need to be serialised to enable deterministic replay [19]. In past work, Courtois *et al.* [9] solve the serialisation problem using a concurrent-read-exclusive-write (CREW) protocol for shared objects, while Russinovich *et al.* [38] propose a repeatable deterministic task scheduler. He have adopted the latter for *PA*, as it outperforms CREW protocols on uniprocessor architectures.

2.2 Synchronisation

Smartphone users enjoy plentiful wireless connectivity over 3G, WiFi, GPRS, *etc.PA* can use any of these networks to synchronise with the replica by transmitting the execution trace. However, wireless connectivity can be costly in terms of energy consumption, and detrimental to battery life. Therefore, we assume that network connectivity may not be always available (*e.g.,* because the device is low on power), and safeguard the execution trace to ensure that attacks which occured while disconnected are eventually discovered.

2.2.1 Loose Synchronisation

We adopt a loose synchronisation strategy between the phone and the cloud to minimise its effects on battery life. Particularly, we do not activate or keep any of the network adaptors from sleeping, but rather attempt to transmit the trace only when the device is awake and connected to the Internet. This can be due to the user performing an action like reading email or surfing the web, when he is also most likely to be attacked (*e.g.,* by receiving a malicious email, or accessing a malicious web site). Alternatively, we also support an extremely loose synchronisation model, where the device synchronises with its replica only when it is recharging. Such a model may be suitable for users with more relaxed security requirements, as attacks can only be detected after synchronising with the server.

2.2.2 Tamper-Evident Secure Storage

Loose synchronisation with the server is ideal for preserving power, but unless we protect the execution trace, an attacker may compromise the phone and disable the synchronisation procedure. Even worse, a capable attacker could modify the execution trace to remove the entries that expose the attack (*e.g.,* a specific read from the network), while keeping the system operational to make it appear as if everything is still running properly.

We defend against such attacks by employing a *secure storage* to detect if someone has tampered with the execution trace. Every block of data written to secure storage is associated with an HMAC code [2], that simultaneously verifies the block's authenticity and integrity. HMAC is a specific type of message authentication code (MAC) that involves a cryptographic hash function in combination with a secret key. We achieve *tamper-evidency* by continuously "rolling" the key used with the HMAC, as we explain below.

Each time an entry along with its HMAC code is written to secure storage, we generate a new key by applying a second cryptographic hash function on the old key (which is completely overwritten). This way an attacker compromising the device, cannot alter old entries already in the execution trace to hide an attack. At worst, attackers can delete entries or block synchronisation, which both count as synchronisation errors.

$$STORE(message + HMAC(key, message))$$
$$key' = HASH(key)$$
$$key = key'$$

Writes to secure storage occur regularly during the operation of *PA*, or can be triggered by a specific event. While the system is running, the data produced by the *tracer* are initially buffered and compressed in the manner described in Section 3.2. When data can no longer be buffered (*e.g.,* because the buffer has been exhausted), or when it is determined that they cannot be further compressed, they are

written to secure storage and a new key is generated. Alternatively, writes to secure storage may be "forced" when certain events occurs, even if additional buffering is possible. For instance, when a network read occurs that could potentially introduce malicious data, we request that the entry describing the network read (as well as the previously buffered entries) are written into secure storage. Different algorithms and strategies that determine the frequency of writes to secure storage can be explored in future work.

Using HMAC is more lightweight than digital signatures, as it requires less processing cycles (and consequently power) and storage. The only requisite is that a secret key is initially shared between the device and the server. Such a key can be established when setting up the device for use with *PA*. The *replayer* authenticates the received data by calculating their HMAC code, and comparing it with the one received.

2.2.3 Synchronisation Errors

An error during synchronisation can be the result of a software bug, or a failed attempt by an attacker to cover his tracks. It can manifest itself as a mismatch in the HMAC code, a corrupted execution trace, or failure to communicate for a long period of time. The true cause of such an error cannot be determined with confidence by the security server, and in any case we lose the ability to further replay execution. Consequently, devices exhibiting such errors are treated as potentially compromised, and the user needs to be notified and his device restored to a clean state (Section 2.5).

2.3 Security Methods

The real power of *PA* lies in the scalability and flexibility in security methods. By replicating smartphone execution in the cloud, we have ample resources for running a combination of security tasks. Moreover, we can apply any detection method that obeys the rule of noninterference For instance, all of the following detection methods are compatible with *PA*'s security model. As a proof of concept, we implemented the first two in the list (Section 3.3) and are currently working on the others.

1. Dynamic analysis in the emulator. We instrument the emulator to perform runtime analysis to detect certain types of zero-day attacks such as buffer-overflows and code-injection attacks [18, 41, 10, 8].
2. AV products in the cloud. We modified a popular open source AV to run in the emulator, and perform periodical file scans. Additionally, on access file scanning can be applied with few modifications to the *replayer*. On access scanning AV intercept file handling system calls and scan the target file before allowing a process to access it. As we already intercept system calls, the *replayer* could be transformed to an on access AV scanner.
3. Memory scanners. We can scan emulator memory for patterns of malicious code directly. Memory scanners are able to detect memory-resident attacks that leave no files behind for AV scanners to detect.
4. System call anomaly detection. Detection methods based solely on the system calls [36, 15], can even be applied directly to the execution trace, without any need for replaying. As a result, system call detection methods are extremely fast.

While, all the techniques we have referred to in this section have been around for some time, execution replay offers great flexibility, even enabling future runtime security solutions to be applied retrospectively. Furthermore, the execution trace can be retained and used for auditing purposes.

2.4 Proxy And Server Location

The location of the security server and the proxy, and who controls them is a policy decision beyond the scope of this paper. For instance, institutions running their own cloud could deploy the proxy and replica in-house. Alternatively, *PA* could be offered as a service by wireless providers, hosting the server on their own cloud. While privacy is important both for companies and individuals, smaller companies and individuals frequently place their data on cloud services offered by providers such as Amazon and Google.

In an extreme scenario, users with strong privacy considerations could run their own replicas on their desktop or notebook, and not use a proxy at all. Doing so gives them full control over their data, but implies a very loose synchronisation model, where the device synchronises with the server only when the device is plugged to the computer, or when they are on the same network (*e.g.,* similarly to Apple's *Time Capsule*).

2.5 User Notification And Recovery

When an attack is detected, *PA* needs to warn the user, so that recovery procedures can be initiated. This is not trivial. Sending an SMS or email message may not work, as a skilled attacker could block such messages. As such, a signalling channel beyond the control of the attacker is needed. The nature of this channel is not very important for this paper, but various options are already available. For instance, we could use special hardware on the phone to have it destroy all data, when it receives a privileged message by the owner or provider (*e.g.,* the "kill pill" message on Blackberry phones [39]). If hardware support is not available, the provider could also simply deny service to the device, which would (hopefully) inform the user that something is wrong.

Compromised devices can be restored to a pristine state using the data held at the replica. Data-loss can be kept at a minimum, as an exact copy of all user data exists in the cloud. Furthermore, using multiple intrusion detection techniques we can accurately detect the moment of the attack, to restore the really last clean state of the system. Unfortunately, recovery over the network cannot be guaranteed, so we adopt an approach similar to current systems such as the iPhone, where the device needs to be plugged-in a PC to be recovered.

2.6 Handling Data Generated On The Device

While we can proxy the data that is already available 'in the network', we cannot do so for data that is generated locally. Examples include key presses, speech, downloads over Bluetooth (and other local connections), and pictures and videos taken with the built-in camera. Keystroke data is typically limited in size. Speech is not very bulky either, but generates a constant stream. We will show in Section 4 that *PA* is able to cope with such data quite well.

Downloads over Bluetooth and other local connections fall into two categories: (a) bulk downloads (*e.g.,* a play list of music files), typically from a user's PC, and (b) incremental downloads (exchange of smaller files, such as ringtones, of-

ten from other mobile devices). Incremental downloads are relatively easy to handle. For bulk downloads, we can save on transmitting the data if we duplicate the transmission from the PC such that it mirrors the data on the replica. However, this is an optimisation that we have not yet applied.

Pictures and videos taken using the device may incur significant overhead in transmission. *PA* caters more to security sensitive environments like corporations and government institutions, where such data are encountered less frequently. Nevertheless, in application domains where such activities are common, users will probably have to disconnect from the server, and only resynchronise when their device is recharging to avoid draining the battery. In the future, we could exploit the increasing trend of users uploading their content to the Internet directly from their devices, to also proxy the uploaded data and make them available to the replica.

3. IMPLEMENTATION

In this section, we discuss a prototype implementation of *PA* for Google's Android system. While it is possible to implement the *tracer* and *replayer* in different ways, the most efficient way is to intercept system calls and signals in the kernel. It is also the most convenient way to influence the scheduling to serialise accesses to shared objects (discussed in 2.1). However, it is hard to maintain such an implementation, as it requires frequent updates to keep it operational with new kernels, and it requires that a new boot image is installed on the device every time the *tracer* is updated. This motivated us to implement *PA*'s prototype in user space.

Our implementation is transparent to applications and the OS, and only requires process tracing functionality, comparable to the one offered by Linux's *ptrace*, which enables us to attach to arbitrary processes, and intercept system calls and signals. Similar interfaces are also support by BSD- and Windows-style OSs used on other devices, such as the iPhone OS and Windows Mobile.

3.1 Recording And Replaying

In this section, we explain the novel aspects of implementing execution recording and replaying on Android.

3.1.1 Starting The Tracer And Everything Else

In UNIX tradition, Android uses the *init* process to start all other processes, including the supporting framework and user applications. The *tracer* itself is also launched by init, before launching any of the processes we wish to trace. *Init* launches the processes that are to be traced using an execution stub. This process serves a twofold purpose: it allows the *tracer* to start tracing the target processes from the first instruction, and it enables us to run processes without tracing them (*e.g.,* debugging and monitoring applications).

Init brings up the *tracer* process first. The *tracer* initialises a FIFO to allow processes that need tracing to contact it. Next, *init* starts the other processes. Rather than starting them directly, we add a level of indirection, which we call the exec stub. So, instead of forking a new thread and using the *exec* system call directly to start the new binary, we fork and run a short stub. The stub writes its process identifier (pid) to the *tracer*'s FIFO (effectively requesting the *tracer* to trace it) and then pauses. Upon reading the pid, the *tracer* attaches to the process to trace it. Finally, the *tracer* removes the pause in the traced process, making the stub resume execution. The stub immediately calls *exec* to start the appropriate binary with the corresponding parameters.

3.1.2 Scheduling And Shared Memory

In Section 2.1, we briefly mentioned that we serialise accesses to shared objects using a modified task scheduler that operates in a deterministic way. Unfortunately, we can only do so with coarse granularity, as we operate entirely in user space. Our scheduling algorithm is quite simple and far from optimal, but sufficient for our purpose, as it is reproducible. Furthermore, it does not require us to log any additional information in the execution trace. It operates by ensuring that no two threads that share a memory object can ever run concurrently. Because the scheduler is triggered by system calls, it can be unfair, and it may theoretically deadlock in the presence of spinlocks. To avoid the latter, we created a spinlock detector that is activated when a task keeps running for more than a predefined period of time. In practice, Android does not use spinlocks as they are wasteful in terms of CPU cycles. Instead, locking is performed using mutexes, which results in a system call in case of contention, and are handled by *PA* in a straightforward way. While the spinlock detector provides the robustness that is required for a production system, so far we have only seen it triggered for contrived test cases.

Modern operating systems also allow processes to directly memory map HW memory. If such memory was to be used for directly reading data from hardware, neither repeatable scheduling nor a traditional CREW protocol could ensure proper serialisation of accesses to that memory. To the best of our knowledge, Android does not use memory in this way. However, it could be a problem in the future in a different hardware/software combination. In that case, we need a modified CREW protocol that will track all reads from such memory to keep execution deterministic. This can be accomplished by making the area inaccessible to the reader, and intercepting all read attempts using the generated page faults. Doing so would be expensive, especially if done from user space. Fortunately, we have had no need for this in our implementation.

3.1.3 Ioctls

I/O control, mostly performed using the *ioctl* system call, is part of the interface between user and kernel space. Programs typically use ioctls to allow userland code to communicate with device drivers. Each request uses a command number which identifies the operation to be performed and in certain cases the receiver. Attempts have been made to apply a formula on this number that would indicate the direction of an operation, as well as the size of the data being transferred. Unfortunately, due to backward compatibility issues and programmer errors actual ioctl numbers do not always follow this convention. Furthermore, Android performs most of its IPC through the kernel using the binder framework [33]. Many of the binder operations actually result in one or more ioctls on the *"/dev/binder"* device. Thus, it is important that we can access the Android kernel source code to check the semantics of the various ioctls being used. Fortunately, smartphones make use of fewer ioctls than PCs, but the procedure is still a tedious one. Our prototype handles about two hundred ioctl commands.

3.2 Execution Trace Compression

One of our primary goals is to minimise transmission costs, which requires minimising the size of the execution trace. Here we discuss the rules we applied to reduce the size of the trace:

- *Record only system calls that introduce nondeterminism.* Phone and replica execute the same instruction stream, so there is no need to record system calls that have identical effects in both (*e.g.,* creating a socket, opening a file, reading from local storage, *etc.*). We also do not record the results of systems call used for IPC between processes, as the mirror processes on the replica will generate the same data.
- *Use a network proxy so that inbound data are not logged in the trace.* The data received by the phone over the network are not directly seen by the replica (*e.g.,* a received email). We introduce a transparent proxy that logs all Internet traffic towards the phone, and makes it available to the security server upon request. This way the phone does not need to waste precious energy to log and transmit them to the replica.
- *Compress data using three algorithms.* First, we encode time related data returned by calls such as *gettimeofday* and *clock_gettime* using delta encoding, replacing the actual time data in the trace with the differences of consecutive values. Second, we employ Huffman encoding to represent frequent values in the trace. For instance, we use a bit to represent a system call that returned zero, another one to indicate that a log entry has been produced by the same thread as the previous entry, *etc.* Finally, we employ general purpose compression using the DEFLATE algorithm (also used by the *gzip* utility).

3.3 Attack Detection Mechanisms

We demonstrate the detection capabilities of the security server by developing two very different detection mechanisms: an anti-virus scanner, and an emulator-based detector that uses dynamic taint analysis.

3.3.1 Virus Scanner

One of the security measures we apply at the server is antivirus scanning. For this purpose, we modified the popular open source anti-virus ClamAV to run in the Android emulator. ClamAV contains more than 500000 signatures for viruses that a user would have to store locally on his phone and update daily. Using *PA*, we perform file scanning on the server where both storage and processing is much cheaper. Moreover, if we wish to further increase detection coverage we may employ multiple scanners at the same time, as suggested by Oberheide *et al.* [30].

3.3.2 Dynamic Taint Analysis

PA can go a lot further than just running multiple antivirus software in the cloud. We modified the Android emulator to apply dynamic taint analysis (DTA) on the replica [10, 27]. DTA is a powerful, but expensive method to detect intrusions. The technique flags all data that arrive from a suspect source (like the network) as tainted. Tainted data are tracked throughout the execution of the system, so that all data the depend on tainted data are also flagged as tainted. For instance, when tainted values are used as source operands in ALU operations or copied, the destination is also tainted. When an attacker exploits a vulnerability (*e.g.,* a buffer overflow, a format string attack, a dangling pointer, *etc.*) to inject and execute arbitrary code, or simply arbitrarily redirect the execution flow of the vulnerable program (*e.g.,* using return-to-libc, or return oriented shellcode), DTA identifies the illegal use of tainted data and raises an alert. For instance, an alert is raised when tainted data are executed, or used as an operand of a *CALL* instruction.

DTA works against a host of exploits, including zero-day attacks, and incurs practically no false positives. Unfortunately, the overhead imposed is very high, making it an impractical solution to deploy on production systems (both PCs and phones). By applying DTA on a smartphone's replica, we manage to hide its overhead from the end user, and concurrently exploit the more powerful hardware in the cloud to accelerate it.

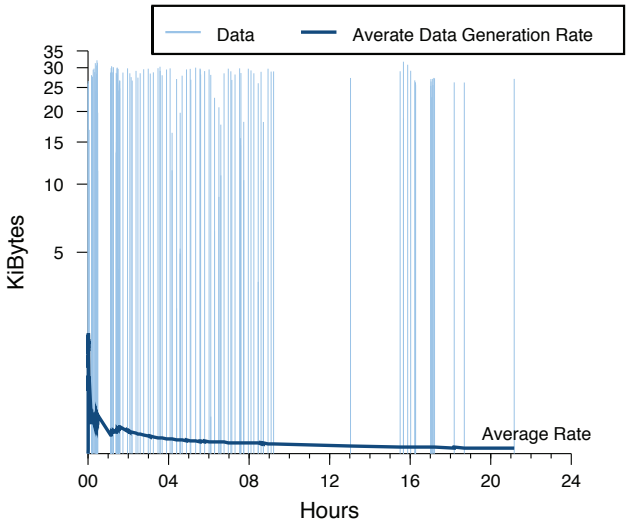


Figure 2: Data generated by *PA* on a user operated HTC G1 for a day.

4. EVALUATION

We evaluate our implementation of *PA* along three axes: the amount of trace data generated during recording, the overhead imposed by the *tracer* on the device, and finally the performance and scalability of the server hosting the replicas. We run the *tracer* on the HTC G1 developer phone, while the *replayer* is hosted on the modified QEMU [1] emulator that comes with the official SDK. We do not perform a security evaluation of our taint analysis implemention on QEMU, as it has been sufficiently demonstrated by [35].

4.1 Data Volume

The volume of data generated by the *tracer* constitutes an important metric, as it directly affects the amount of energy required to transmit the trace log to the server, and the storage space needed to store it on the device when disconnected from the server. Additionally, the upload bandwidth available to smartphone users (usually a few hundred Kbps) is a scarce resource, as it is frequently much less than the

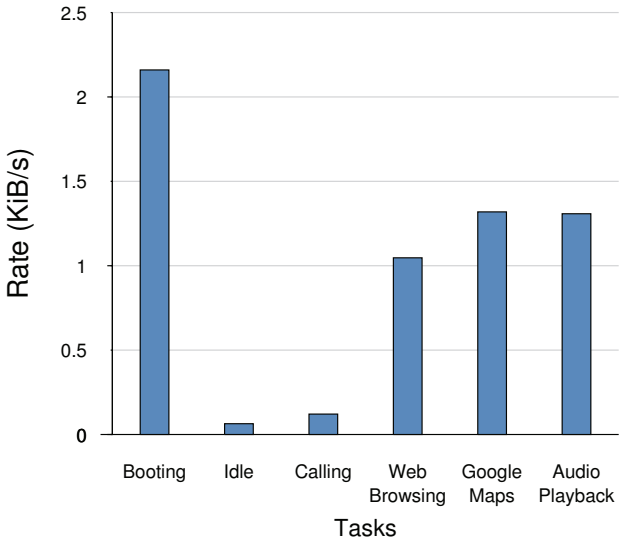


Figure 3: Average data generation rate, when performing various tasks.

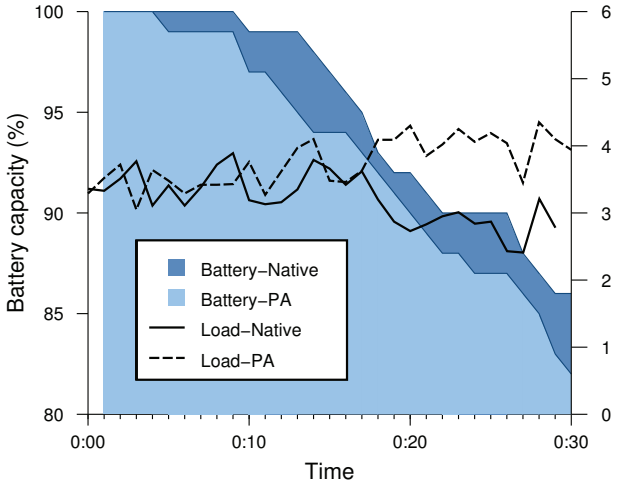


Figure 4: Battery level and CPU load average while browsing on the HTC G1 developer phone. We draw two independent experiments, where we browse URLs from [7] natively and under *PA*.

available download bandwidth.

Our traces collected from actual users using their phones show, not surprisingly perhaps, that mobile devices are mostly idle, or used for voice calls. A plot of the amount of data generated by *PA* over time is shown in Figure 2. Meanwhile, Figure 3 shows that the data generated when the device is idle or the user is making a call is negligible, with an average of 64B/s and 121B/s respectively. Even when performing more intensive tasks, such as browsing or listening to music, the *tracer* generates less than 2KiB/s. For instance, 5 hours[2] of audio playback would generate about 22.5MB of trace data. Transmitting such an amount of data solely over 3G may burden users with excessive costs, specially when operators cap their bandwidth and charge extra for data transfers over the cap, but it can be easily stored locally on the smartphone (devices already offer relatively large amounts of storage; *e.g.,* the iPhone 4 offers 32GB of storage) until a WiFi connection is available. Taking into account that many users spend most of their time at home or at the office, it is very likely that WiFi connectivity will be frequently available to synchronize with the server.

4.2 Overhead

PA imposes two types of overhead on the phone. First, it consumes additional CPU cycles and thus incurs a performance overhead. Second, it consumes more power because of the increased CPU usage and the transmission of the execution trace to the server. To quantify these costs, we monitored the device's CPU load average, and battery level, while randomly browsing URLs from [7]. We performed this task natively as well as under *PA*, and show the results in Figure 4.

Figure 4 confirms that *PA* increases the CPU load on the device. In particular, the mean CPU load during this experiment was about 15% higher when using *PA*. The use of compression and encryption is somewhat costly in terms of processing, but the amount of data we generate does not seem to justify for such a divergence. The figure also shows how battery capacity drops in time while browsing. As expected power is consumed faster when using *PA*. When idle or in light use, the additional battery consumption is minimal, but heavyweight tasks, such as browsing may consume up to 30% more energy.

However, most of this overhead seems to be due to the additional CPU cycles consumed by the user space *tracer*. We confirmed this by way of an experiment where we only transmit the trace data corresponding to the browsing activity (using SSL as the tracer would do), and found that the device did not report *any* drop in battery level. We investigate the cause behind this increase in CPU load and battery consumption, and discuss our findings in 4.4.

4.3 Server Scalability

Chun *et al.* [6] has shown that simply moving computation from a smartphone to faster hardware such as a PC, even when running on an emulator, can increase performance up to 11.8 times. While we cannot replay execution faster than it is recorded, the significant difference in processing power between smartphones and PCs enables us to host multiple replicas on each security server.

We corroborate our assumption by measuring the number of phone replicas that can be run concurrently on various hardware configurations. Each replica was run on the Android Qemu-based emulator, executing the same task as the original. It is also in-sync with the replayed device, *i.e.,* the replica has to wait for trace data from the device. While running the replicas, we did not introduce any detection mechanism or instrumentation, which represents an optimal scenario for this experiment. The results are shown in Figure 5, where we draw the number of replicas that can be run under these conditions using different hardware configurations. Particularly, we used a dual-core (x2 2.26GHz P8400) HP HDX18 notebook with 4GB of RAM, a four-core (x4 2.40GHz Q6600) desktop PC with 8GB of RAM, and a high-memory extra-large instance on Amazon's Elastic

[2]Typical battery life when browsing or reproducing audio can range from 3 to 8 hours depending on the device.

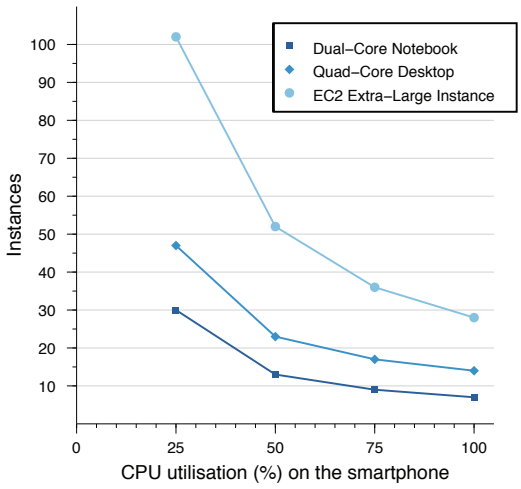


Figure 5: Number of replica instances that can be run on a server without delay. As CPU utilisation increases on the phone, fewer replicas can be executing in sync with the phone.

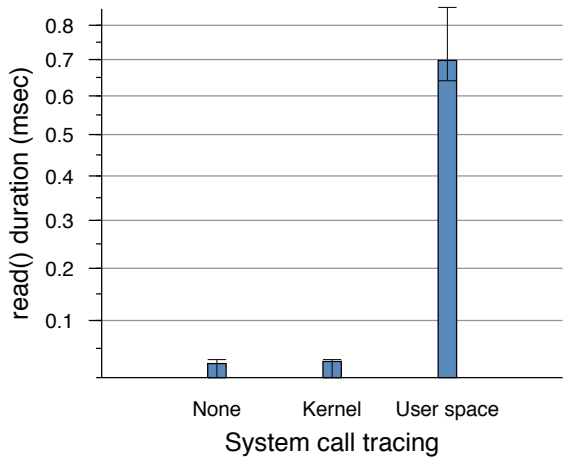


Figure 6: The time it takes to read 4Kbytes of data from */dev/urandom* natively, and when tracing.

Function	Time Spent %
ptrace()	% 33.63
waitpid()	% 32.68
deflate_slow()	% 7.62
pread64()	% 6.78
mcount_interval()	% 2.84
event_handler_run()	% 2.15

Table 1: Top executing functions in the *tracer*.

Cloud (EC2) service with 68.4GB or RAM. *When running in the EC2 cloud, we were able to concurrently run more than 100 replicas of devices exhibiting an average 25% CPU utilisation.* Utilisation is a key factor, since it determines the number of replicas that can be run without delays, as computation is relatively expensive when running under the emulator.

Determining the average CPU utilisation of smartphones in a realistic scenario is not a trivial task, and we are not aware of any preexisting studies on the subject. Nevertheless, we can look at the intensity of different tasks commonly performed on these devices. For instance, on the HTC G1 developer phone we measured 90%-100% CPU utilisation when running a game, 20%-25% when reproducing audio, 30%-100% when browsing, and finally 0%-5% when the phone is idle. We can intuitively argue that smartphones remain idle or run non-interactive tasks like listening to music most of the time. In the opposite case, battery is drained quickly by the CPU (when running intensive tasks such as browsing or gaming), the display, and various device sensors (GPS, accelerometer, *etc.*).

We already mentioned that the results presented in Figure 5 present an optimal scenario, as no security mechanism is applied. *PA*'s scalability actually depends on the type and number of mechanisms introduced at the server. For instance, previous work that implemented DTA for the x86 architecture using the Qemu emulator reported a x2-x2.5 slowdown compared with execution under Qemu alone. We obtained similar results implementing DTA for the ARM architecture using Android's Qemu-based emulator. As such, we estimate that if DTA is applied on every replica, we would be able to run roughly half of the instances reported in Figure 5 without any delay. Finally, we have tested our scheme on Amazon's EC2 cloud service to demonstrate the scalability of our approach. In practice, organisations that are willing to invest in smartphone security, will most probably host their own security servers as well as proxies to ensure that privacy sensitive data remain within the organisation, and to reduce costs[3].

4.4 Overhead Imposed By Ptrace

We mentioned earlier that we observed an increase in CPU load and consequently battery consumption under *PA* that was unexpected. We investigated further by profiling the *tracer* to identify its "expensive" functions, and list the top five functions in Table 1. We see that compression (performed by *deflate_slow*) consumes only 7.62% of the *tracer*'s execution time, and no cryptographic function is even reported in the top results. On the other hand, a bit more than 65% is spent in *ptrace* and *waitpid*. The latter is called continuously to retrieve events from the kernel. Every time a traced process enters or exits a system call, it is blocked and such an event is delivered to the *tracer* through *waitpid*. Additionally, we use ptrace at least once for every event to retrieve additional information (*e.g.,* the system call number). These two calls cause a large number of context switches between the *tracer*, traced processes, and the kernel, and incur the larger part of the overhead we observe. Similarly, *pread64* is used to copy data from the memory of traced processes (such as data returned by a system call).

We are confident that moving event reception and the initial part of event handling of *PA* in the kernel, would greatly improve performance. This is supported by what we see in Figure 6. Even when tracing a single system call like *read*, using *ptrace* incurs a huge overhead when compared with

[3]Products that offer data proxying are already available for devices like BlackBerry smartphones [3].

native execution. On the contrary, tracing the same system call, including copying the returned data, within the kernel imposes no observable overhead. Future work on *PA* will focus on moving part of the implementation in the kernel.

5. RELATED WORK

The idea of decoupling security from execution has been explored previously in a different context. Malkhi *et al.* [22] explored the execution of Java applets on a remote server as a way to protect end hosts. The code is executed at the remote server instead of the end host, and the design focuses on transparently linking the end host browser to the remotely executing applet. Although similar at the conceptual level, one major difference is that *PA* is *replicating* rather than *moving* the actual execution, and the interaction with the operating environment is more intense and requires additional engineering.

Ripley [40] is another system that proposes the replication of an application in a server to automatically preserve its integrity. Unlike *PA*, it focuses on distributed web 2.0 applications, and particularly AJAX based applications. Attacks are detected by comparing the results of the replica with the client's. A discrepancy indicates an attack, so Ripley is in fact investing on the two executions deviating. Furthermore, it does not apply to a broad range attacks like *PA*, and it is not transparent to the application.

The idea of off-loading execution from smartphones to the cloud was first proposed in CloneCloud [6]. The main focus of this work is the acceleration of CPU intensive and low interaction applications. While the authors recognize its potential use for decoupling security from phones, they do not investigate the effects of disconnected operation on security (*i.e.,* the need for secure storage), nor do they investigate the cost of replication for the phone and the server. Finally, CloneCloud is not always transparent to applications.

Decoupling security from smartphones was first explored in SmartSiren [4], albeit with a more traditional anti-virus file-scanning security model in mind. As such, synchronisation and replay is less of an issue compared to *PA*. Oberheide *et al.* [30] explore a design that is similar to SmartSiren, focusing more on the scale and complexity of the cloud backend for supporting mobile phone file scanning, and sketching out some of the design challenges in terms of synchronisation. Some of these challenges are common in the design of *PA*, and we show that such a design is feasible and useful. However, both these approaches can only protect against a limited set of attack vectors.

Other work on smartphone security includes VirusMeter by Liu *et al.* [21]. This work also identifies that traditional defences do not perform as well on smartphones due their limited resources. They propose using power consumption levels to identify potentially malicious software operating on a smartphone. Their solution uses very little resources, but it may incur false positives. Enck *et al.* address the issue of malicious applications downloaded on smartphones with Kirin [13]. They propose a system that can automatically analyse applications submitted to application stores (*e.g.,* Google's *Marketplace* and Apple's *Apple Store*) for potentially malicious behaviour. Kirin is orthogonal to our system, and could in fact be used in combination.

Our architecture also bears some similarities to BugNet [26] which consists of a memory-backed FIFO queue effectively decoupled from the monitored applications, but with data periodically flushed to the replica rather than to disk. We store significantly less information than BugNet, as the identical replica contains most of the necessary state.

6. CONCLUSION

In this paper, we have discussed a new model for protecting mobile phones. These devices are increasingly complex, increasingly vulnerable, and increasingly attractive targets for attackers because of their broad application domain. The need for strong protection is apparent, preferably using multiple and diverse attack detection measures. Our security model performs attack detection on a remote server in the cloud where the execution of the software on the phone is mirrored in a virtual machine. In principle, there is no limit on the number of attack detection techniques that we can apply in parallel. Rather than running the security measures locally, the phone records a minimal execution trace, and transmits it to the security server, which faithfully replays the original execution.

The evaluation of a user space implementation of our architecture *Paranoid Android*, shows that transmission overhead can be kept well below 2.5KiBps even during periods of high activity (browsing, audio playback), and to virtually nothing during idle periods. Battery life is reduced by about 30%, but we show that it can be significantly improved by implementing the *tracer* within the kernel. We conclude that our architecture is suitable for protection of mobile phones. Moreover, it offers more comprehensive security than possible with alternative models.

Acknowledgments

This work has been supported by the European Commission through projects FP7-ICT-216026-WOMBAT and FP7-ICT-257007 SYSSEC. Also, with the support of the Prevention, Preparedness and Consequence Management of Terrorism and other Security-related Risks Programme European Commission - Directorate-General Home Affairs. This publication reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

7. REFERENCES

[1] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proc. of USENIX'05*, April 2005.

[2] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Proc. of Crypto'96*, pages 1–15, August 1996.

[3] BlackBerry, Inc. BlackBerry Enterprise Server. http://na.blackberry.com/eng/services/business/server/full/.

[4] J. Cheng, S. H. Wong, H. Yang, and S. Lu. SmartSiren: virus detection and alert for smartphones. In *Proc. of MobiSys'07*, pages 258–271, June 2007.

[5] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proc. of USENIX'08*, pages 1–14, June 2008.

[6] B.-G. Chun and P. Maniatis. Augmented smartphone applications through clone cloud execution. In *Proc. of HotOS XII*, May 2009.

[7] A. T. W. I. company. Top 500 global sites. http://www.alexa.com/topsites.

[8] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worm epidemics. In *Proc. of SOSP'05*, October 2005.

[9] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, 1971.

[10] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[11] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proc. of OSDI'02*, pages 211–224, December 2002.

[12] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proc. of VEE '08*, pages 121–130, March 2008.

[13] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proc. of CCS*, pages 235–245, 2009.

[14] F-Secure. "sexy view" trojan on symbian s60 3rd edition. http://www.f-secure.com/weblog/archives/00001609.html, February 2008.

[15] J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *Proc of NDSS'04*, February 2004.

[16] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *Proc. of OSDI*, 2008.

[17] L. Hatton. Reexamining the fault density component size connection. *Software, IEEE*, 14(2):89–97, 1997.

[18] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Proc. of the 11th USENIX Security Symposium*, pages 191–206, August 2002.

[19] T. Leblanc and J. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 36(4):471–482, 1987.

[20] G. Legg. The bluejacking, bluesnarfing, bluebugging blues: Bluetooth faces perception of vulnerability. http://www.wirelessnetdesignline.com/192200279?printableArticle=true, August 2005.

[21] L. Liu, G. Yan, X. Zhang, and S. Chen. VirusMeter: Preventing your cellphone from spies. In *Proc. of RAID*, pages 244–264, 2009.

[22] D. Malkhi and M. K. Reiter. Secure execution of java applets using a remote playground. *IEEE Trans. Softw. Eng.*, 26(12):1197–1209, 2000.

[23] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Proc. of ASPLOS '09*, pages 73–84, March 2009.

[24] H. Moore. Cracking the iPhone (part 1). http://blog.metasploit.com/2007/10/cracking-iphone-part-1.html, October 2007.

[25] R. Naraine. Google Android vulnerable to drive-by browser exploit. *http://blogs.zdnet.com/security/?p=2067*, October 2008.

[26] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. *SIGARCH Comput. Archit. News*, 33(2):284–295, 2005.

[27] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of NDSS'05*, February 2005.

[28] Niacin and Dre. The iPhone/iTouch tif exploit is now officially released. Available at http://toc2rta.com/?q=node/23, October 2007.

[29] J. Oberheide, E. Cooke, and F. Jahanian. CloudAV: N-version antivirus in the network cloud. In *Proc. of the 17th USENIX Security Symposium*, San Jose, CA, July 2008.

[30] J. Oberheide, K. Veeraraghavan, E. Cooke, J. Flinn, and F. Jahanian. Virtualized in-cloud security services for mobile devices. In *Proc. of MobiVirt '08*, pages 31–35, June 2008.

[31] oCERT. CVE-2009-0475: OpenCORE insufficient boundary checking during MP3 decoding. http://www.ocert.org/advisories/ocert-2009-002.html, January 2009.

[32] A. Ozment and S. E. Schechter. Milk or wine: Does software security improve with age? In *Proc. of the 15th USENIX Security Symposium*, July 2006.

[33] I. PalmSource. OpenBinder. http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html, 2005.

[34] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: Zero-day protection for smartphones using the cloud. Technical report, Vrije Universiteit Amsterdam, 2010.

[35] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. In *Proc. of ACM EuroSys*, April 2006.

[36] N. Provos. Improving host security with system call policies. In *Proc. of the 12th USENIX Security Symposium*, August 2003.

[37] M. Ronsse and K. De Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

[38] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proc. of PLDI '96*, pages 258–266, May 1996.

[39] V3.co.uk. BlackBerry 'kill pill' vital for IT security. http://www.v3.co.uk/vnunet/news/2159105/blackberry-kill-pill-vital.

[40] K. Vikram, A. Prateek, and B. Livshits. Ripley: automatically securing web 2.0 applications through replicated execution. In *Proc. of CCS*, pages 173–186, 2009.

[41] J. Xu and N. Nakka. Defeating memory corruption attacks via pointer taintedness detection. In *Proc. of DSN '05*, pages 378–387, June 2005.

[42] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. *SIGARCH Comput. Archit. News*, 31(2):122–135, 2003.

Exploiting Smart-Phone USB Connectivity For Fun And Profit

Zhaohui Wang
Department of Computer Science
George Mason University, Fairfax, VA
zwange@gmu.edu

Angelos Stavrou
Department of Computer Science
George Mason University, Fairfax, VA
astavrou@gmu.edu

ABSTRACT

The Universal Serial Bus (USB) connection has become the de-facto standard for both charging and data transfers for smart phone devices including Google's Android and Apple's iPhone. To further enhance their functionality, smart phones are equipped with programmable USB hardware and open source operating systems that empower them to alter the default behavior of the end-to-end USB communications. Unfortunately, these new capabilities coupled with the inherent trust that users place on the USB physical connectivity and the lack of any protection mechanisms render USB a insecure link, prone to exploitation. To demonstrate this new avenue of exploitation, we introduce novel attack strategies that exploit the functional capabilities of the USB physical link. In addition, we detail how a sophisticated adversary who has under his control one of the connected devices can subvert the other. This includes attacks where a compromised smart phone poses as a Human Interface Device (HID) and sends keystrokes in order to control the victim host. Moreover, we explain how to boot a smart phone device into USB host mode and take over another phone using a specially crafted cable. Finally, we point out the underlying reasons behind USB exploits and propose potential defense mechanisms that would limit or even prevent such USB borne attacks.

1. INTRODUCTION

Recent advances in the hardware capabilities of the mobile hand-held devices have fostered the development of open source operating systems for mobile phones. These new generation of smart phones such as iPhone and Google Android phone are powerful enough to accomplish most of the tasks that previously required a personal computer. Indeed, this newly acquired computing power gave rise to plethora of applications that attempt to leverage the new hardware. This includes Internet browsing, email, GPS navigation, messaging, and custom applications to name a few. In addition, the ubiquitous use and the wide-spread adoption of Universal Serial Bus (USB) [7] led the phone device manufacturers to equip the majority of third-generation phones with USB ports. In fact USB is currently employed as a means of charging, communicating, and synchronizing the contents of the phone with computers and other phones. Moreover, to support an open programming model that allow third party developers to contribute their applications, these new devices come with an extended set of features. These features enable them use the USB interface to perform more complex functions including data and application synchronization.

In this paper, we assume the role of an adversary and study the new threats that stem from the use of USB interface to connect, synchronize, and program the mobile device. Unlike the network and bluetooth communications for mobile devices that have defense mechanisms in place, USB traffic is not authenticated, filtered, or vetted. For example, to establish bluetooth connectivity, the user is required to enter a password to establish connection between unpaired devices. Moreover, all cellular and wireless communication connections and packets are inspected by stateful firewall or intrusion detection systems. On the other hand, USB connections are overlooked both by the users and by the defenses and are assumed as a trusted communication channel. This inherent trust is rooted in the belief that physical proximity implies trust. To debunk that myth, we explain how software vulnerabilities in today's mobile devices can spread through the USB interface and affect both the USB device and the host that is connected to.

This new threat vector creates the potential for malware to take over a smart phone device when the device is connected via standard USB to an infected computer and vice-versa. In practice, a malicious host can abuse the USB connection to unlock and flash the software of the phone bypassing all software and hardware defenses. Reversely, we show how a malicious smart phone device can take over a computer by posing as a Human Interface Device (HID) such as a keyboard or a mouse among others. Additionally, we detail how an adversary can abuse the inherent USB mounting and synchronization capabilities to run malicious code on the host computer. To make matters worse, we illustrate attacks that can empower an infected smart phone to connect and take over another smart phone by placing its USB connection into the USB-host mode. Current smart phone devices run full-fledged mobile operating systems. These mobile operating systems provide a programmable interface to control the existing USB ports thus empowering them to launch attacks against desktop computers rather than merely acting as a USB storage device.

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

In addition, in most cases, smart phones connect to one or more desktop systems for file backup, data synchronization in addition to charging the battery. The strong coupling relationship between the device and the desktop system makes either side vulnerable to attacks that exploit this tight and trusted coupling when the other is compromised. Most end-users have little or no knowledge about the system running on the phone. To make matters worse, the device vendors lock the phone by default disallowing the end-users from having full access to the device. In the meantime, locking the device does not prevent or even deter an experienced adversaries or malicious code from attacking the mobile devices. Moreover, the USB functionality on the smart phone can be programmed to play the role of a USB host and drive other "peripheral" devices. This can be leveraged to attack other USB devices including smart phones.

Furthermore, currently USB-borne attacks are not considered as a problem: most of the current mobile security research focus on malicious applications [21, 19, 11]. This includes mobile phone rootkits such as Cloaker [12] and others [10]. In addition, drive-by downloads from untrusted sources, execution of foreign code, leaking of sensitive information, corruption and file integrity are just a few among the current threats that the mobile phones face. Unlike previous research, we focus on the new avenues of infection that go beyond the regular software vulnerabilities spread via the cellular or network connections. Our aim is to study and model new possible mechanisms available to mobile malware through exploiting the technical capabilities of the mobile device. We don't devise new exploit payloads but rather expose new avenues of automatic and stealthy exploitation. Any existing or future exploits can take advantage of this new ways to spread and propagate between devices.

The main contributions of this paper are summarized as follows:

- We are the first to study attacks that take advantage of the USB interface connectivity and utilize it as an avenue of exploitation. To that end, we show how malicious code can leverage USB as a new infection vector for propagation and self-replication.
- We present examples of attacks for three basic connectivity scenarios: Phone-to-Computer, Computer-to-Phone, and Phone-to-Phone. Also, we provide a detailed description of the required steps for attacks in each scenario. We demonstrate that it's enough for an adversary controlling one end of the USB connecting ends to infect the other end.
- Finally, we discuss the potential defenses based on common limitations of such USB-borne attacks.

The rest of this paper is organized as follows: Section 2 introduces the motivation and background about contemporary smart phone devices. The threat model and the description of the new USB attacks are presented in Section 3. We discuss the underlying limitations of attacks as well as potential defenses in Section 4. Section 5 presents security related research on mobile device and mobile operating system security and Section 6 concludes this paper.

2. MOTIVATION & BACKGROUND

2.1 Motivation

Currently, USB connections are inherently trusted and assumed secure by the users. This can be partly attributed to the physical proximity of the device and the desktop system and the fact that, in most cases, the user owns both systems. However, as we show, this trust can be easily abused by a malicious adversary. For instance, in a typical usage scenario, an unsuspected user connects the smart phone device to her computer to charge its battery and to synchronize the two devices including her contact list, calendar and media content. All of these tasks are performed automatically either completely transparently to the user or with minimal user interaction: the simple press of a mouse click upon connecting the USB cable. To make matters worse, the computer is completely unaware of the type of the device that is connected to the USB port. As we elaborate later, this observation can be exploited by a sophisticated adversary to launch attacks against the desktop system. Furthermore, there are no mechanisms to authenticate the validity of the device that attempts to communicate with the host. This lack of authentication allows the connecting device to disguise and report itself as another type of USB device, abusing the ubiquitous nature operating system.

Traditionally, a smart phone device is connected to the host as a peripheral USB device. Being controlled by the host, the device is more prone to be taken over by a compromised computer. However, the potential attack surface is much wider: the USB creates a bidirectional communication channel, permitting, in theory, exploits to traverse both directions. New generation phones are equipped with complete operating systems which make them as powerful as a desktop system. These recent hardware advancements enables them to perform attacks that are far beyond their previous computational and software capabilities. Additionally, unlike desktop computers and servers that do not change their physical location, phones are mobile. This empowers them to potentially communicate to an even larger number of un-infected devices across a wider range of administrative domains. For example, a smart phone left unattended for a few minutes can be completely subverted and become an point of infection to other devices and computers. Lastly, because USB-borne attacks have not been seen before, there are no defenses in place to prevent them from taking place or even detect them.

In the meantime, the lack of deployed USB defenses or detection mechanisms empowers the attacks to remain stealthy. Currently, the only instance of USB-borne threats is flash drive viruses spreading from USB files. However, the new smart phones are capable of accomplishing a much more powerful and widespread propagation of malfease. The propagation that can be caused by this new infection vector goes beyond viruses that are passively hidden in traditional USB storage devices. The above observations motivate our study of this new infection vector that is spurred by the new technology trends, as well as propose potential defenses.

In the next section, we briefly introduce hardware and software background information necessary to understand the technical details behind the new USB attacks. Even though we implemented the attacks using specific devices, the threats that the USB connectivity raise apply in general to all smart phone devices.

Devices	USB interface types
iPhone/iTouch	Apple Proprietary 5-pin wide USB
Motorola Droid and other Android based	Micro USB AB
HTC Windows CE-Based	Micro HTC ExtUSB with 11-pin connector
Old Nokia models	Pop-Port connector
Google's Nexus One	Micro USB AB

Table 1: USB interfaces of various mobile devices.

Device Name	Description	Device Type	VendorID	ProductID	Service Name	Driver Filename	Serial Number
SE Flash OMAP3430 MI	Motorola Flash Interface	Vendor Specific	22b8	41e0	MotDev	motodrv.sys	
SE Flash OMAP3430 MI	USB Composite Device	Unknown	22b8	41e1	usbccgp	usbccgp.sys	
Palm Handheld	Palm Handheld	Vendor Specific	0830	0061	PalmUSBD	PalmUSBD.sys	PalmSN12345678
Nexus One	Google, Inc.Nexus One USB Device	Unknown	18d1	4e11	usbccgp	usbccgp.sys	HT9CNP804091
Nexus One	USB Mass Storage Device	Mass Storage	18d1	4e11	USBSTOR	USBSTOR.SYS	
Nexus One	Android ADB Interface	Vendor Specific	18d1	4e11	WinUSB	WinUSB.sys	
Nexus One	Gadget Serial	CDC Data	18d1	4e11	usbser	usbser.sys	
Nexus One	Nexus One	Vendor Specific	18d1	4e11			
Motorola A855	Motorola A855 USB Device	Unknown	22b8	41db	usbccgp	usbccgp.sys	040388000E00C01D
Motorola A855	USB Mass Storage Device	Mass Storage	22b8	41db	USBSTOR	USBSTOR.SYS	
Motorola A855	Mot Composite ADB Interface	Vendor Specific	22b8	41db	androidusb	motoandroid.sys	

Figure 1: The logical communication channels of the composite USB Device as they appear in Windows XP systems.

2.2 Background

Here, we discuss the background information and the specific devices employed in our experiments. In 2008, Google and Open Handset Alliance launched Android Platform[1] for mobile devices. Google's Android is a comprehensive software framework for mobile communication devices (i.e., smart phones, PDAs). The Android framework is an full operating system including system library files, middleware, and a set of key applications.

Nowadays, most smart phones are equipped with a Mini USB or Micro USB interface for PC to phone connectivity. This USB interface provides the physical link for the synchronization of contacts and calendar data. Table 1 gives the different USB interfaces with different devices.From the operating system point of view, all Android driven devices contain more than one interface descriptor, which is known as a composite USB device. This physical link can be multiplexed: with a single physical USB interface, the device can act as multiple devices simultaneously as long as they comply with the USB specification.

For our experiments, the device is Google's Nexus One. The operating system is Android 2.1 (codename *eclair*). While Google's website [5] lists the specifications from a marketing point of view, Table 2 lists the hardware modules of the device from the operating system's point of view: the second column is the internal device driver names of the different modules. Table 3 provides the MTD (Memory Technology Device) device partition layout , whereas MTD is the Linux abstraction layer between the hardware-specific device drivers and higher-level applications. How fast we can flash the device depends on the size of the storage each specific device equipped with. In addition to the NAND device storage, Google's Nexus One uses a 4GB sd card as external storage. This works as separated device in the Android operating system and can be mounted as a USB mass storage device to the desktop system. We will leverage this hardware design to launch the Phone-to-Computer attacks. In the manufacture state, the Google's Nexus One has only two logical USB interfaces by default, one is the USB mass storage while the other is the Android ADB Interface. By modifying the kernel source code with corresponding kernel compilation options, we enabled other hidden USB interfaces in the kernel, show in Figure 1.

3. NOVEL INFECTION VECTORS

3.1 Threat Model

To establish basic communication, the both end of the USB connection are connected via off-the-shelf USB cables. In our threat model, we assume an adversary that is already in control of one end of the USB connection. This is true for all our three attack scenarios. For instance, in the Phone-to-Computer attacking scenario, the phone is fully under the control of the adversary. Moreover, we assume that the attacker can manipulate any component of the device, ranging from applications to programmable hardware components. The victim, in this case the desktop system, is assumed to have a basic set of device drivers that come with the installation of the operating system and support Human Interface Device (HID) installation. Note that this is not an additional step required to be accomplished by the adversary. In the case of Computer-to-Phone infection, we assume the desktop system is compromised. Put it differently, we assume that the adversary has already placed malicious software that runs alongside with the regular legitimate software. The phone is considered intact and in the default manufacturer state. We only focus on how the compromised desktop system could infect the phone and propagate malware while connected through USB to the device. How the desktop system became comprised is beyond the scope of this paper. Such exploitation can be accomplished via traditional browser exploitation, email phishing, or buffer overflow.

For Phone-to-Phone attacks, the attacking device is manipulated to take over the innocent victim device. Beyond the full control of the mobile operating system of the at-

Modules	Hardware
CPU	Qualcomm QSX8250 1Ghz
Mother board	Qualcomm Mobile Station Modem (MSM) SoC
RAM	512 MB
ROM	512 MB , partitioned as boot/system/userdata/cache and radio
External Storage	4GB micro SD
Audio Processor	Msm_qdsp6 onboard processor
Camera	5 MegaPixels Sensor_s5k3e2fx
Wifi+BlueTooth+FM	Boardcom BCM 4329, 802.11a/b/g/n
Touch Screen Input	Msm_ts touchscreen controller, capella
Vibrator	Msm_vibrator on board vibrator
Digital Compass	AK8973 compass

Table 2: Google's Nexus One Hardware Modules.

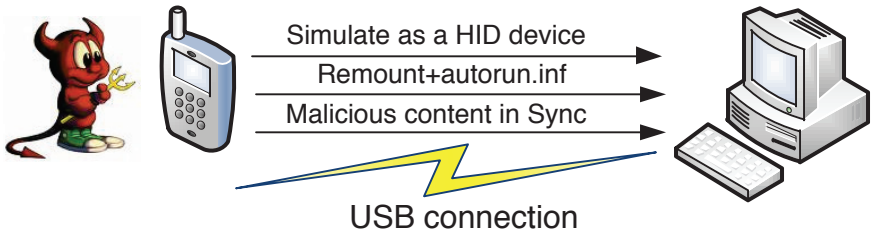


Figure 2: The Phone-to-Computer Attacks over the USB Connection.

tacking device, the adversary also has to craft a special USB cable. This cable is used to place the malicious device into USB host-mode and establish a connection to the the target phone device. We explain the necessary USB cable modifications in Section 3.4. Having established a thread model and listed our assumptions, we detail the steps to accomplish USB-borne attacks in the following sections.

3.2 Phone-to-Computer Attacks

Upon connection, USB becomes a bidirectional communication channel between the host (normally a desktop system) and the peripheral device. The established belief that only the master device (i.e the host computer) is potentially capable of taking over the slave device (i.e. the smart phone) is incorrect. Indeed, an attacker can launch attacks and transfer malicious programs from a USB peripheral to the machine that acts as a host. Launching attacks against the connected desktop system is a new emerging avenue of exploitation that can be used to spread malware. We demonstrate this new infection vector by focusing on two general classes of attacks which have not been introduced previously.

The first class takes advantage of the fact that smart phones have open source operating systems and can pose as Human Interface Device (HID) peripherals (also called gadgets) and connect to the computer. This new functionality can be leveraged by an sophisticated adversary to cause more damage than traditional passive USB devices. The second class of attacks harnesses the capability of the phone to be automatically mounted as a USB device and automatically run content. The process of a USB device being mounted is not a threat on its own. Even having the possible malware hidden in sd card partition in the device and mounted on the computer as a USB stick is not a novel attack. However, being able to identify the operating system on the other side of the USB connection and prepare an attack payload *selectively* is a new attack capability. This is because the phone can arbitrarily control and repeat this mount and unmount operation within the device.

To demonstrate first class of attacks, we developed a special USB gadget driver in addition to existing USB composite interface on the Android Linux kernel using the *USB Gadget API for Linux*[8]. The UGAL framework helped us implement a simple USB Human Interface Driver (HID) functionality (i.e. device driver) and the glue code between the various kernel APIs. Using the code provided in: "drivers/usb/gadget/composite.c", we created our own gadget driver as an additional composite USB interface. This driver simulates a USB keyboard device. We can also simulate a USB mouse device sending pre-programmed input command to the desktop system. Therefore, it is straightforward to pose as a normal USB mouse or keyboard device and send predefined command stealthily to simulate malicious interactive user activities. To verify this functionality, in our controlled experiments, we send keycode sequences to perform non-fatal operations and show how such a manipulated device can cause damages In particular, we simulated a Dell USB keyboard (vendorID=413C, productID=2105) sending "CTRL+ESC" key combination and "U" and "Enter" key sequence to reboot the machine. Notice that this only requires USB connection and can gain the "current user" privilege on the desktop system. With the additional local or remote exploit sent as payload, the malware can escalate the privilege and gain full access of the desktop system.

Another class of attacks are content exploitations. Such attacks take advantage of media content to exploit vulnerable softwares that exist in the victim system. These attacks are not new and have been known for quite some time (e.g. PDF and Flash exploits). However, we show

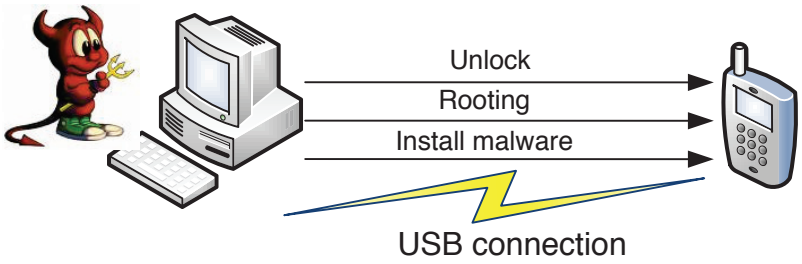


Figure 3: The Computer-to-Phone Attacks over the USB Connection.

a new way to accomplish these attacks using the USB connection. In Android devices, in addition to the NAND device, an sd card works as external storage. This separated device can be mounted as a USB mass storage device to the desktop system. There are system-wide options for the user to set:1, connecting only for battery charging;2, allowing NAND ROM device available to the desktop system via USB Android Debugging Bridge driver (*adb*);3, allowing sd card device available to the desktop system as a USB mass-storage device. If the last option is set, the sd card device is *automatically* mounted by generic USB mass-storage driver in major commodity operating systems by default bypassing any restrictions. We leverage this platform-specific observation to implement the basic attack against the desktop system. Our malicious program drops an *autorun.inf* and the *calc.exe* to the sd card partition. The next time when the user want to transfer files (e.g. movie, photo, mp3 file etc), once the sd card is mounted as a partition, the calc.exe will be executed in our default configuration Windows XP system [2].

Moreover, unlike the traditional passive USB stick devices, the CPU powered phone as a USB peripheral device promotes the attacks in a more intelligent manner. As a starting point, we (the attacker) wrote the malware on the phone monitoring the USB connectivity. Once the phone is connected to a desktop system, we probe and identify the operating system by looking at the URB (USB Requesting Block) ID in the USB packets. By doing this, we differentiate the targeted system and avoid brute force approaches. After the target system is being identified, using the computational power on the phone, we enumerate the available vulnerabilities and change the attacking payload with multiple runs with different content. For example, in our controlled experiments, the targeted desktop system is a Windows XP SP3 with a vulnerable version Adobe PDF software and fully updated JPG parse engine. Our proof-of-concept malware on the phone will compose the *autorun.inf* upon detecting it is a Windows, and launch Windows Picture and Fax Viewer program to view the special crafted JPG file and the PDF program to view the malicious PDF file we dropped. We observed the expected result that the malicious logic in the crafted PDF file was executed and the Windows system is compromised. We acknowledge that this depends on malware-writer's knowledge on contemporary vulnerabilities. However, the CPU equipped phone device as a gadget can help malware-writers generate composite malware and highly infectious code, to achieve higher successful ratio.

For iPhone devices, the strong coupling between iTunes software and iPhone devices makes such Phone-to-Computer attacks even simpler. Once the iPhone connected to the desktop system, the iPhone/iPod Service installed by iTunes will detect the device and launch iTunes. iTunes will scan the media content on the device and make them available in the iTunes. Since the attacker has the full control of the device, it can drop any specially crafted media file (e.g. jpg, pdf, mp3, mov etc) to exploit the corresponding processing engine.

3.3 Computer-to-Phone Attacks

In this section, we detail the steps required to take over a smart phone device when its connected via the USB port to a computer. A closer look into the attacking process reveals that it can be decomposed into a sequence of operations. The phone is not unlocked and in manufacture out-of-box state in terms of installed software. This is usually true for most of the end-users. To mount the attack, we take advantage of the open source program *fastboot* which can manipulate the boot-loader of the Android phone devices. By issuing the command *fastboot oem unlock*, the device will display a warning page and once we click "yes", it is officially unlocked and the manufacture warranty also is voided. However, this is far from being inconspicuous and requires user input. To achieve fully automation, we crafted a small program to simulate the clicking of yes action. We do so by sending the touchscreen input event with the corresponding touchscreen coordinators need be pressed directly via the USB connection. Upon completion of the unlocking process, we can replace the system images. This means that all software including kernel, libraries, utility binaries, and applications are now under our control. The second step is to do a full system dump from device, so that we can exfiltrate all the programs and user information. This can be used for phishing purposes in addition to creating a backup of the applications to prevent the user from noticing any changes in the device.

The entire unlocking and flashing process takes 4 mins 5 seconds on our device and may vary for different devices due to different content sizes. To be more specific, we flash the recovery partition using a third party modified recovery image which provide the functionality that can do a whole NAND file system backup based on the partition information in Table 3. Such backup covers boot partition, system partition, userdata partition, and a hash checksum. We disassemble this boot partition dump *boot.img* to a raw kernel zimage binary file and corresponding ram-disk file. The boot.img file is composed with the kernel in zimage format, the compressed ram-disk in gzip format, and the paddings. The overall layout of the boot.img file is listed as follows: 0x0-0x7ff: File Magic:"Android!",kernel size in bytes, kernel physical loading address, ram-disk size in bytes, ram-disk

Dev	Size	Name	Range	Erasesize
mtd0:	0x000e0000 896KB	misc	0x000003ee0000-0x000003fc0000	0x00020000
mtd1:	0x00500000 5MB	recovery	0x000004240000-0x000004740000	0x00020000
mtd2:	0x00280000 2.5MB	boot	0x000004740000-0x0000049c0000	0x00020000
mtd3:	0x09100000 145MB	system	0x0000049c0000-0x00000dac0000	0x00020000
mtd4:	0x05f00000 95MB	cache	0x00000dac0000-0x0000139c0000	0x00020000
mtd5:	0x0c440000 196.24MB	userdata	0x0000139c0000-0x00001fe00000	0x00020000

Table 3: Google's Nexus One NAND Partition Layout.

physical loading address, product name, kernel command line options (512bytes), timestamp, sha1 hash. 0x800:4K page aligned kernel zimage with zero trailing paddings after that is the ram-disk which also 4K page aligned and zero padded. The last part is a second optional kernel for testing and do not normally appear in device. We use such knowledge to repack the boot.img file which includes malicious code.

Google maintains regular release and updates for Android system, and all the boot.img files are publicly available as well as other system files. The user may update the boot.img on it's own and we can not assume it has the same boot.img as Google's released standard ones. For a particular victim device, we do not have the prior knowledge about this boundary information between the kernel and the ram-disk. Since the magic string of gzip file is 0x1F8B, we use 0x000000001F8B which is the trailing padding zeroes plus the gzip magic string as the identification of the start ram-disk content, and rewrite them to separate files. After we get the ram-disk file, we unpack it and get direct access to *init.rc* file. This file is parsed by *init* program which is also the first process of the system. It sets up the basic environment for the system and then launches critical system daemon processes and services. The *init* binary and *init.rc* include Android specific system features (e.g some global system properties are defined and parsed here) and are critical to the entire system. Until now, we assumed direct access to all the resources to insert our malicious logic into the system. Initially, we bind the *adbd* daemon process with root permission by changing the *adbd* parameters *init.rc* file. This will provide root shell access to the whole system when we launch *adb* connection from our desktop system as a attack vector. Afterwards, we use the command in *init.rc* to remount system partition as read-only or we can run"(mount yaffs2 mtd@system /system ro remount,mount rootfs rootfs / ro remount)" to achive full filesystem privileges regardless of the system settings. Then, we add new command in *init.rc* file to launch the malicious program as a system service which will be pushed into the system as a separate step so that it is persistent and still running after phone reboot or battery outage. It is worth mentioning that this makes the malicious program persistent at bootup and is agnostic to the malware code itself. If the malicious binary is removed, such automated initialization will fail. The path need to match the corresponding path of the binary.

After performing the aforementioned modifications, we repack the *boot.img* from the modified sources and flash it back to boot partition on the device. The repack process is straightforward: we compress the modified ram-disk files and directory structures into a single *ramdisk.cpio.gz* file. We then combine it with the kernel and kernel command line options by *mkbootimg* program which is available in Android repository. The flashing process merely takes 2 seconds for a 2560KB *boot.img* file by issuing command *fastboot flash boot boot.img* where *fastboot* is a program having the minimal functionality of maintaining the device in boot-loader mode (e.g. updating partitions of the device). This program is available for Windows, Linux, and Mac OSX. After all the above steps, we have gained full control of the victim device and prepared automated launching of the malicious code. We reboot the phone back to normal mode from boot-loader mode and push our malicious binary to the system partition by *adb push evilprog /system/xbin* and change the permission for execution. The detailed malicious action that this evil binary can do is beyond the scope of this paper. For proof-of-concept demonstration purposes, we wrote a program for collecting the device information and send them to a pre-configured internal collection server stealthily over TCP/IP via cellular data network or wireless network whichever available. This program is cross-compiled against Android's bionic C libraries with arm-eabi toolchains. Some more developed and foreseen real attacks are discussed in Section 4. Note that this program is written in C and executed as the ARM ELF binary at the system utility level which is lower than Davik Java virtual machine and bypass all Android's permission checks for application at JVM [14]. Our server successfully collected the device information sent by the program, which includes the serial number of the device, the kernel version and a list of installed applications.

As we mentioned earlier in this section, all the above logic and operation sequences are programmed as a malicious daemon running on the desktop system. The complete process takes 300 seconds, which corresponds to the sum of every steps.

3.4 Phone-to-Phone Attacks

The inherent mobility and programmability of the third-generation smart phones gave rise to a new type of insider attack. The phone is fully capable of assuming the role of a computer host by setting its USB port to be a USB Hub. This type of attack is similar to the attacks described in Section 3.3. For phone-to-phone attacks, a malicious user connects a subverted device to a victim device and then take over it stealthily. This can happen, for instance, when the victim device is left unattended. In this section, we show how to perform a phone-to-phone attack via a single USB interface as the infection vector. The key capability is to enable the USB host mode on one device, a Motorola Droid in our case, which first time provides the ability of controlling a Android device from another Android device. The rest of the attack is similar to the one described in Section 3.3. When the manipulated Motorola Droid device

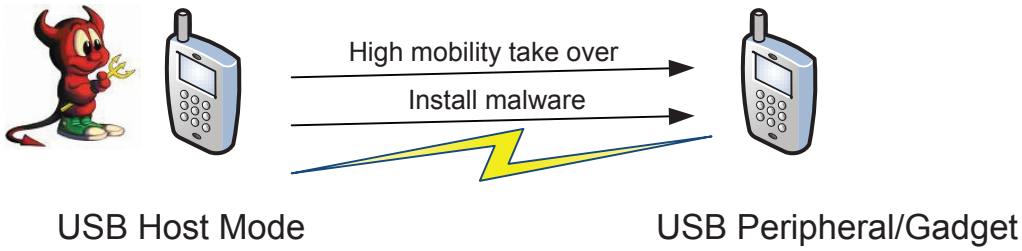


Figure 4: The Phone-to-Phone Attacks over the USB Connection.

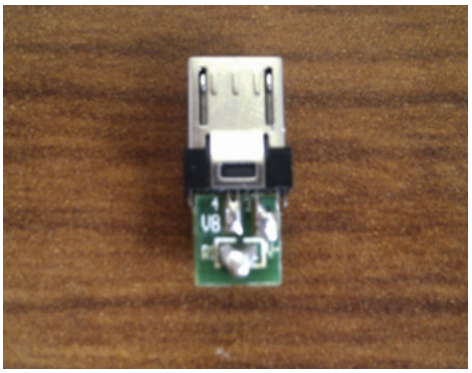


Figure 5: The Micro B USB Connector Dongle.

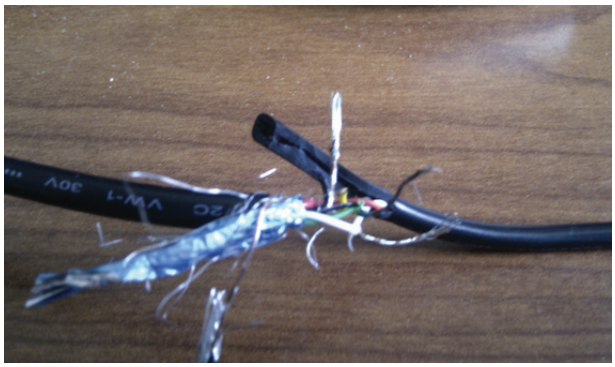


Figure 6: The Crafted USB Cable for Phone-to-Phone Attacks.

connected to another device, the malicious daemon will send pre-programmed command and the victim device will treat it as from a normal desktop system.

For our purposes, we leverage the advanced USB chip in recent released Google Nexus One by HTC and Motorola Droid devices and enable the device's USB host mode capabilities. In regular operation, the phone devices only act as peripheral devices at the USB protocol level. The desktop system will send the first USB packet and initiate the USB connection link. We instead enable the USB OTG (On-the-Go) driver in the device with such hardware support, and flip a normal smart phone device as the USB host. To be more specific, both Nexus One's Qualcomm QSX8250 chipset and Motorola Droid's Texas Instruments OMAP3430 chipset support USB OTG specification [9]. Our experiment on Google Nexus One device failed due to limited SoC depended kernel code support for Qualcomm QSX8250 chipset. However, the OMAP series chipset integrated with the Philips ISP1301 USB OTG transceiver has more mature code in the kernel source. By checking the following kernel compilation options, we can enable the OTG software.

```
CONFIG_ARCH_OMAP_OTG=y
CONFIG_USB_OTG=y
CONFIG_USB_MUSB_OTG=y
CONFIG_USB_OTG_UTILS=y
```

After we activate the kernel driver, we need the specially crafted USB connectors and cable to trigger the USB host mode of the USB OTG device and connect other peripheral devices. By soldering the 4th pin and 5pin of the micro USB connector from a car charger, we changed a micro B connector to a micro A connector, to identify itself as a host side connector. Unfortunately, most off- the-shelf product do not specify it is a A connector or a B connector. Figure 5 shows the micro B dongle we had to solder to achieve our goal. To place the device in the USB hub mode, we have to perform a hard reboot while the micro B connector is inserted in the Droid USB interface. Moreover, we have to unplug the micro-dongle as soon as the Motorola logo disappears as the Droid logo appears. This forces the hardware initialization process to identify the USB hardware in the host mode. After the system boots up, we can verify that the USB is in host mode by running the following command "cat /sys/devices/platform/musb_hdrc/mode". If the output of the command is "a_host" then we are in host mode. Notice that we need to enable the wireless connectivity and use secure shell connection for shell access because the USB interface is in host mode and thus traditional *adb* shell access over USB is disabled.

To connect other peripheral devices, in our case a victim phone, we make the special USB cable with both end micro USB by cutting two cables and put two micro connector in a single cable by soldering the same color together. Our additional experiments shows the device can support additional USB-to-Serial converter but for USB flash driver devices, we have to use external USB power hub to supply additional power to the Vcc line. Figure 6 depicts a snapshot of the cable we made with the micro USB connectors at both ends. It is worth mentioning here that due to the requirement that the D+ and D- must be twisted for synchronization purposes, we can only break the cable within a limited distance for soldering.

Another important aspect of the attack is that the peripheral device driver must be compiled in the host mode device. To limit unnecessary code, most of the non-required kernel options and device drivers are turned off by manufacture configuration. We performed our experiments using a Motorola Droid to attack a Nexus One phone. The generic

USB hub driver on the Droid kernel is compiled as part of the Linux Kernel. The final step is compiling the user level program against the Android system libraries. *adb* provides the ability of controlling a Android device from another Android device. The rest of the attack is similar to the one described in Section 3.3 where the host is replaced with the Droid device. When the malicious Motorola Droid device connects to the victim device, the malicious daemon will send the pre-programmed command over the USB and the victim device will treat it similarly as it did for the host computer.

4. DISCUSSION

Our attacks are primarily implemented on the Android framework because of its open source nature and the ease that we can demonstrate and detail our results making them reproducible. However, we posit that attacks that abuse the USB physical link and hardware programmability exist also for other mobile phone platforms such as the Apple iPhone OS, Microsoft Windows CE and Symbian OS. Moreover, there are scenarios where the described classes of attacks are easier to be accomplished on other platforms. Taking iPhone OS as an example, an adversary can take advantage of the default music play functionality that iTunes software offers to craft malware media files and "synchronize" them with the connected computer. In addition, antivirus products normally scan the external storage in the device which appears as a flash drive from the operating system's view. However, such scans are based on well-known file formats and none of them can scan the internal ROM or raw data stored in the hand-held devices, to the best knowledge of the authors. This represents a clear defense gap.

The common theme behind the USB attacks is the established belief that physical cable connectivity can be inherently trusted and that peripherals are not capable of abusing the USB connection. To protect the end-point devices, there is a need to shed that belief. Instead we have to focus on how to establish trust that is not implicit but explicit and puts the human on the loop. Therefore, a possible defense strategy is to authenticate the USB connection establishment phase and communications using similar techniques that were developed for Bluetooth devices. This will give a visual input to the user and will allow her to verify that a device that attempts to connect as a peripheral is indeed allowed to connect. Moreover, there is a need to identify and communicate to the user the type of the USB device that attempts to connect as a peripheral. This will prevent attacks that pretend to be HID devices and connect without any user interaction.

Unfortunately, attacks that exploit the USB while the victim device is in "slave" mode are more difficult to thwart because some of the functionality is required to control the "slave" device. However, smart phone vendors can try to filter and vet the USB communications using a USB firewall. Similar to network firewall, this USB firewall will inspect all USB packets coming to the device and check the content based on platform-specific rules preventing attacks that replay key-strokes via the USB bypassing the user-input.

In the meantime, we can protect the smart phone system by performing a full backup. This is an easy solution and feasible for most mobile devices. Indeed, the internal ROM storage is relatively limited on smart phones, 512 MB in our case. Using a program that runs on the phone, we can easily dump the entire filesystem using prior knowledge about the partition information to a back-end desktop systems or even external sdcard storage. Note that such backup is the complete filesystem, which includes boot partition and kernel binaries. If the backup is performed from a clean state, a simple revert can defeat all persistent malware even rootkits. However, restoring the phone to a pristine state might lead to loss of user personalization data and thus, it can only act as an emergency measure and not a full-proof or even user friendly solution.

5. RELATED WORK

Platform-specific attacks and defenses: The presentation "Understanding Android's Security Framework" [14] presents a high-level overview of the mechanisms required to develop secure applications within the Android development framework. The tutorial contains the basics of building an Android application. However, the described interfaces must be carefully secured to defend against general malfeasance. They showed how Android's security model aims to provide mechanisms for requisite protection of applications and critical smart phone functionality and present a number of "best practices" for secure application development within the environment. However, authors in [21] showed that this is not enough and that new semantically rich and application-centric policies have to be defined and enforced for Android. Moreover, in [19] the authors show how to establish trust and measure the integrity of application on mobile phone systems. At Black Hat 2009 [11] the authors focus mainly focus on the application security on Android platform. Unlike software, Android devices do not all come from one place. The open nature of the platform allows for proprietary extensions and changes. The proposed extensions can help or could interfere with security. Shabtai *et al.* [23, 24] assess the security mechanisms incorporated in Google's new Android framework. The authors provide a list of security mechanisms which can be incorporated to harden the security of Android. They also make some recommendations on the efficacy and priorities of various security mechanisms. They've seen attacks and current threats against mobile phones in the listed subsystems. Some of the vulnerabilities exist already in the wild while some of them are imminent to be wildly spread in the near future[3]. TaintDroid [13], is designed to expose how user-permitted applications actually access and use private or sensitive data. This includes location, phone numbers and even SIM card identifiers, and to notify users in realtime. Their findings suggest that Android, and other phone operating systems, need to do more to monitor what third-party applications are doing when running in smart phones.

Rookits on mobile devices : Cloaker [12] is a non-persistent rootkit which does not alter any part of the host operating system (OS) code or data, thereby achieving immunity to all existing rootkit detection techniques which perform integrity, behavior and signature checks of the host OS. Cloaker leverage the ARM architecture design to remain hidden from currently deployed rootkit detection techniques, so it's architecture specific but OS independent. [10] uses three example rootkits to show that smart phones are just as vulnerable to rootkits as desktop operating systems. However, the ubiquity of smart phones and the unique interfaces that they expose, such as voice, GPS and battery, make the social consequences of rootkits particularly devastating.

Power Drain Attacks: In [22, 16] the authors study malware that aims to deplete the power resources on the mobile devices. The provided solutions involve changes in the GSM telephony infrastructure. Their work shows that attacks were mainly carried out through the MMS/SMS interfaces on the device. In addition, in [18] the authors show that applications can simply overuse the WiFi, Bluetooth or display of the device and eventually cause a denial of service attack. VirusMeter [17] modeled the power consumption and detect the malware based on power abnormality. However the use of linear regression model with static weights for devices' relative rate of battery consumption is a totally non-scalable approach [20].

Stealthy Video & Audio Surveillance: Xu et al [26] describe a novel attack which stealthily captures video using the on-board camera found on smart phones. Their algorithm covertly records video according to the phone usage and uses a compression algorithm to store the video on disk. This file can later be transferred to the attacker. These attacks are very realistic and go easily unnoticed to the user of the device. However, they do not propose any solutions.

Text Messages Attacks: In addition to the research mentioned in power drain attacks which exploits SMS/MMS functionality [22], Traynor et al. [15], show how specially crafted message packets could compromise a city wide GSM infrastructure, with mitigating mechanism proposed in [25]. Researchers at McAfee Avert Labs have observed examples of SMS (short message service) phishing (also known as SMiShing), which seems to be on the rise [6]. One example is malware that uses the text-messaging APIs to send fake messages to people on the contact list.

Buffer overflows: Buffer overflows also plague mobile devices. The presentation on hacking Windows Mobile [4] at Xcon 2005 talked shell code development advice as well as sample code. Recent emerging threats show that such exploitations are targeting web browsers and other potentially exploitable software like adobe pdf view application in the mobile OSes.

6. CONCLUSIONS

In this paper, we introduced several new types of attack vectors that attempt to take advantage of the inherent trust that users place on the physical USB connectivity between a smart phone and their computer. Such attacks became feasible because of the newly introduced hardware and software capabilities of the third-generation smart phones. The use of open source operating systems and programmable USB ports empower a sophisticated adversary to exploit the unprotected physical USB connection between devices. Indeed, we describe how an adversary that has under his control one of the connected devices can subvert the other. Moreover, we show that by crafting a USB cable capable of putting a subverted smart phone to host mode, we are able to exploit other phone devices.

Although we performed our experiments and USB attacks on Android platforms, which by itself includes devices from many manufacturers, we explain how these attacks can be generalized to other third-generation smart phone devices including Apple's iPhone. Finally, we discuss the underlying reasons why USB attacks are a successful avenue of exploitation and propagation of malware and we propose potential defense mechanisms that would limit or even prevent such attacks from taking place in the future.

7. ACKNOWLEDGEMENTS

We would like to thank Nelson Nazzica, Quan Jia, Meixing Le and Jiang Wang from the Center for Secure Information Systems at George Mason University for their comments on our early draft. We also thank the anonymous ACSAC reviewers for their constructive comments. This work was supported in part by US National Science Foundation (NSF) grant CNS-TC 0915291 and a research fund from Google Inc. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the NSF.

8. REFERENCES

[1] Android. http://developer.android.com/.

[2] Autoplay in windows xp: Automatically detect and react to new devices on a system. http://msdn.microsoft.com/en-us/magazine/cc301341.aspx.

[3] Dark side arises for phone apps. http://online.wsj.com/article/SB10001424052748703340904575284532175834088.html?mod=WSJ_newsreel_technology.

[4] Hacking windows ce. http://www.phrack.org/issues.html?issue=63&id=6.

[5] Nexus one features and specifications. http://www.google.com/phone/static/en_US-nexusone_tech_specs.html.

[6] Sms phishing, records system and method. http://www.f-secure.com/weblog/archives/archive-042007.html.

[7] Usb 2.0 specification. http://www.usb.org.

[8] Usb gadget api for linux. http://www.kernel.org/doc/htmldocs/gadget.html.

[9] Usb on-the-go. http://www.usb.org/developers/onthego/.

[10] Bickford, J., O'Hare, R., Baliga, A., Ganapathy, V., and Iftode, L. Rootkits on smart phones: attacks, implications and opportunities. In *HotMobile '10: Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications* (New York, NY, USA, 2010), ACM, pp. 49–54.

[11] Burns, J. Mobile application security on android. In *Black Hat '09* (2009), Black Hat USA.

[12] David, F. M., Chan, E. M., Carlyle, J. C., and Campbell, R. H. Cloaker: Hardware supported rootkit concealment. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 296–310.

[13] Enck, W., Gilbert, P., gon Chun, B., Jung, L. P. C. J., McDaniel, P., and Sheth, A. N. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI '10: Proceedings of the 9th symposium on Operating systems design and implementation* (New York, NY, USA, 2010), ACM, pp. 255–270.

[14] Enck, W., and McDaniel, P. Understanding android's security framework. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security* (New York, NY, USA, 2008), ACM, pp. 552–561.

[15] Enck, W., Traynor, P., McDaniel, P., and La Porta, T. Exploiting open functionality in

sms-capable cellular networks. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security* (New York, NY, USA, 2005), ACM, pp. 393–404.

[16] Kim, H., Smith, J., and Shin, K. G. Detecting energy-greedy anomalies and mobile malware variants. In *MobiSys '08: Proceeding of the 6th international conference on Mobile systems, applications, and services* (New York, NY, USA, 2008), ACM, pp. 239–252.

[17] Liu, L., Yan, G., Zhang, X., and Chen, S. Virusmeter: Preventing your cellphone from spies. In *RAID '09: Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection* (Berlin, Heidelberg, 2009), Springer-Verlag, pp. 244–264.

[18] Moyers, B. R., Dunning, J. P., Marchany, R. C., and Tront, J. G. Effects of wi-fi and bluetooth battery exhaustion attacks on mobile devices. In *HICSS '10: Proceedings of the 2010 43rd Hawaii International Conference on System Sciences* (Washington, DC, USA, 2010), IEEE Computer Society, pp. 1–9.

[19] Muthukumaran, D., Sawani, A., Schiffman, J., Jung, B. M., and Jaeger, T. Measuring integrity on mobile phone systems. In *SACMAT '08: Proceedings of the 13th ACM symposium on Access control models and technologies* (New York, NY, USA, 2008), ACM, pp. 155–164.

[20] Nash, D. C., Martin, T. L., Ha, D. S., and Hsiao, M. S. Towards an intrusion detection system for battery exhaustion attacks on mobile computing devices. In *PERCOMW '05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications Workshops* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 141–145.

[21] Ongtang, M., Mclaughlin, S., Enck, W., and Mcdaniel, P. Semantically rich application-centric security in android. In *In ACSAC '09: Annual Computer Security Applications Conference* (2009).

[22] Radmilo Racic, D. M., and Chen, H. Exploiting mms vulnerabilities to stealthily exhaust mobile phoneŠs battery. In *In SecureComm 06* (2006), SECURECOMM, pp. 1–10.

[23] Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., and Dolev, S. Google android: A state-of-the-art review of security mechanisms. *CoRR abs/0912.5101* (2009).

[24] Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., Dolev, S., and Glezer, C. Google android: A comprehensive security assessment. *IEEE Security and Privacy 8* (2010), 35–44.

[25] Traynor, P., Enck, W., McDaniel, P., and Porta, T. L. Mitigating attacks on open functionality in sms-capable cellular networks. *IEEE/ACM Trans. Netw. 17*, 1 (2009), 40–53.

[26] Xu, N., Zhang, F., Luo, Y., Jia, W., Xuan, D., and Teng, J. Stealthy video capturer: a new video-based spyware in 3g smartphones. In *WiSec '09: Proceedings of the second ACM conference on Wireless network security* (New York, NY, USA, 2009), ACM, pp. 69–78.

Defending DSSS-based Broadcast Communication against Insider Jammers via Delayed Seed-Disclosure*

An Liu, Peng Ning, Huaiyu Dai, Yao Liu
North Carolina State University
{aliu3, pning, huaiyu_dai, yliu20}@ncsu.edu

Cliff Wang
U.S. Army Research Office
cliff.wang@us.army.mil

ABSTRACT

Spread spectrum techniques such as Direct Sequence Spread Spectrum (DSSS) and Frequency Hopping (FH) have been commonly used for anti-jamming wireless communication. However, traditional spread spectrum techniques require that sender and receivers share a common secret in order to agree upon, for example, a common hopping sequence (in FH) or a common spreading code sequence (in DSSS). Such a requirement prevents these techniques from being effective for anti-jamming *broadcast* communication, where a jammer may learn the key from a compromised receiver and then disrupt the wireless communication. In this paper, we develop a novel Delayed Seed-Disclosure DSSS (DSD-DSSS) scheme for efficient anti-jamming broadcast communication. DSD-DSSS achieves its anti-jamming capability through randomly generating the spreading code sequence for each message using a random seed and delaying the disclosure of the seed at the end of the message. We also develop an effective protection mechanism for seed disclosure using content-based code subset selection. DSD-DSSS is superior to all previous attempts for anti-jamming spread spectrum broadcast communication without shared keys. In particular, even if a jammer possesses real-time online analysis capability to launch reactive jamming attacks, DSD-DSSS can still defeat the jamming attacks with a very high probability. We evaluate DSD-DSSS through both theoretical analysis and a prototype implementation based on GNU Radio; our evaluation results demonstrate that DSD-DSSS is practical and have superior security properties.

1. INTRODUCTION

Spread spectrum wireless communication techniques, including Direct Sequence Spread Spectrum (DSSS) and Frequency Hopping (FH), have been commonly used for anti-jamming wireless communication [6]. However, with traditional spread spectrum techniques, it is necessary for senders and receivers to share a secret key to determine, for example, the frequency hopping patterns in FH and the Pseudo-Noise (PN) codes in DSSS. Otherwise, sender and receivers cannot establish anti-jamming communication. Moreover, if a jammer knows the secret key, she can replicate the secret hopping pattern or PN codes and jam the wireless communication.

The above limitations of traditional anti-jamming techniques have motivated a series of recent research. To remove the dependency on pre-shared keys, an Uncoordinated Frequency Hopping (UFH) technique was recently developed to allow two nodes to establish a common secret for future FH communication in presence of a jammer [19]. This approach was latter enhanced in [7,18,20] with various coding techniques to provide more efficiency and robustness during key establishment.

Besides UFH and its variations [7,18–20], two other approaches were recently investigated to enable jamming-resistant broadcast communication *without* shared keys [2,15]. BBC was proposed to achieve broadcast communication by encoding data into "indelible marks" (e.g., short pulses) placed in "locations" (e.g., time slots), which can be decoded by any receiver [2,3]. However, the decoding process in BBC is inherently sequential (i.e., the decoding of the next bit depends on the decoded values of the previous bits). Though it works with short pulses in the time domain, the method cannot be extended to DSSS or FH without significantly increasing the decoding cost.

An Uncoordinated DSSS (UDSSS) approach was recently developed [15], which avoids jamming by randomly selecting the spreading code sequence for each message from a public pool of code sequences. UDSSS allows a receiver to quickly identify the right code sequence by having each code sequence uniquely identified by the first few codes. However, if the jammer has enough computational power, using the same property, she can find the correct sequence before the sender finishes the transmission and jam the remaining transmission. Thus, UDSSS is vulnerable to *reactive jamming attacks*, where the jammer can analyze the first part of transmitted signal and jam the rest accordingly. To mitigate such attacks, a solution similar to ours was proposed in [14]. The basic idea is to spread each message using a key and transmit the key later using UDSSS. To mitigate the reactive jamming attack against the key transmission, UDSSS can trade the resilience for efficiency by setting a larger spreading code sequence set size. On the contrary, our paper tries to provide an alternative solution achieving both resilience and efficiency.

In this paper, we develop Delayed Seed-Disclosure DSSS (DSD-DSSS), which provides efficient and robust anti-jamming broadcast communication without suffering from reactive jamming attacks. The basic idea is two-fold: First, the code sequence used to spread each message is randomly generated based on a random seed only known to the sender. Second, the sender discloses the random seed

*This work is supported by the National Science Foundation under grants CNS-1016260 and CAREER-0447761, and by the Army Research Office under staff research grant W911NF-04-D-0003. The contents of this paper do not necessarily reflect the position or the policies of the U.S. Government.

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

at the end of the message, after the message body has been transmitted. A receiver buffers the received message; it can decode the random seed and regenerate the spreading code using the seed to despread the buffered message. A jammer may certainly try the same. However, when the jammer recovers the random seed and spreading code sequence, all reachable receivers have already received the message; it is too late for the jammer to do any damage.

We also develop a *content-based code subset selection* scheme to protect the random seed disclosure. We use the content of the seed to give some advantage to normal receivers over reactive jammers. This scheme allows a normal receiver, who starts decoding a message after fully receiving the message, to quickly decode the random seed. In contrast, a jammer, who needs to disrupt the message while it is being transmitted, has to consider many more choices.

Our contribution in this paper is as follows. First, we develop the novel DSD-DSSS scheme to provide efficient anti-jamming broadcast communication without shared keys. Our approach is superior to all previous solutions. Second, we develop a content-based code subset selection method to provide effective protection of seed disclosure in DSD-DSSS. Third, we give in-depth performance and security analysis for these techniques in presence of various forms of jammers, including reactive jammers that possess real-time online analysis capabilities. Our analysis demonstrates that our approach provides effective defense against jamming attacks. Finally, we implement a prototype of DSD-DSSS using USRPs and GNU Radio to demonstrate its feasibility.

The remainder of the paper is organized as follows. Section 2 describes background information about DSSS. Section 3 presents our assumptions and the threat model. Section 4 proposes DSD-DSSS and analyzes its anti-jamming capability and performance overheads. Section 5 gives the content-based code subset selection scheme and analyzes its effectiveness. Section 6 shows the implementation and experimental evaluation of DSD-DSSS. Section 7 describes related work, and Section 8 concludes this paper.

2. BACKGROUND

Spread spectrum techniques, including DSSS and FH, use a much larger bandwidth than necessary for communications [6, 16]. Such bandwidth expansion is realized through a spreading code *independent* of the data sequence. In DSSS, each data bit is spread (multiplied) by a wide-band code sequence (i.e., the *chipping sequence*). The spreading code is typically pseudo-random, commonly referred to as *Pseudo-Noise (PN) code*, rendering the transmitted signal noise-like to all except for the intended receivers, which possess the code to despread the signal and recover the information.

Figure 1 shows the typical steps in DSSS communication. Given a message to be transmitted, typically encoded with Error Correction Code (ECC), the sender first spreads the message by multiplying it with a spreading code. Each bit in the message is then converted to a sequence of chips[1] according to the spreading code. The result is modulated, up-converted to the carrier frequency, and launched on the channel. At the receiver, the distorted signal is first down-converted to baseband, demodulated through a matched filter, and then despread by a synchronized copy of the spreading code. The synchronization includes both bit time synchronization and chip time synchronization, guaranteeing that receivers know when to apply which spreading code in order to get the original data. Alternatively, a DSSS system may modulate the signal before the spreading step at sender, and despread and demodulate the received signal at receiver.

[1]To distinguish between bits in the original message and those in the spread result, following the convention of spread spectrum communication, we call the "shorter bits" in the spread result as *chips*.

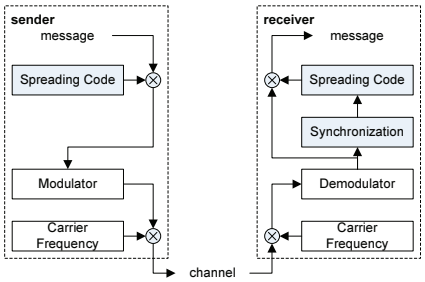


Figure 1: DSSS communication system

The performance of DSSS communication depends on the design of spreading codes. A spreading code $c(t)$ typically consists of a sequence of l chips $c_1, c_2, ..., c_l$, each with value 1 or -1 and duration of T_c, where l is the code length and T_c is chip duration. Assume the bit duration is T_b. The number of chips per bit $l = T_b/T_c$ approximates the bandwidth expansion factor and the processing gain. Two functions characterize spread code: *auto-correlation* and *cross-correlation*. Auto-correlation describes the similarity between a code and its shifted value. Good auto-correlation property means the similarity between a code and its shifted value is low; it is desired for multi-path rejection and synchronization. Cross-correlation of two spreading codes describes the similarity between these two codes; low cross-correlation is desired for multiuser communications.

3. ASSUMPTIONS AND THREAT MODEL

In this paper, we consider the protection of DSSS-based wireless *broadcast* communication against jamming attacks (i.e., one sender and multiple receivers). We adopt the same DSSS communication framework as illustrated in Figure 1. However, the sender and receivers use different strategies to decide what spreading codes to use during broadcast communication. That is, our approach customizes the generation and selection of spreading codes during DSSS communication to defend against insider jamming attacks.

We assume that the jammers' transmission power is bounded. In other words, a jammer cannot jam the transmission of a message unless she knows the spreading codes used for sending the message. For simplicity, we assume the length of each broadcast message is fixed. Such an assumption can be easily removed, for example, by using a message length field.

Threat Model: We assume that the attacker may compromise some receivers, and as a result, can exploit any secret they possess to jam the communication from the sender to the other receivers. We assume intelligent jammers that are aware of our schemes. In addition to injecting random noises, the jammer may also modify or inject meaningful messages to disrupt the broadcast communication.

The jammers may possess high computational capability to perform real-time online analysis of intercepted signal. However, due to the nature of DSSS communication (i.e., each bit data is transmitted through a sequence of pseudo-random chips), it takes time for a jammer to parse the chips for any 1-bit data to determine the spreading code. When the jammer receives enough chips for a given bit to guess the spreading code with a high probability, most of the chips have already been transmitted. Jamming the remaining chips will not have high impact on the reception of this bit. Thus, we assume that if a jammer does not know the spreading code for any 1-bit data, she cannot jam its transmission based on real-time

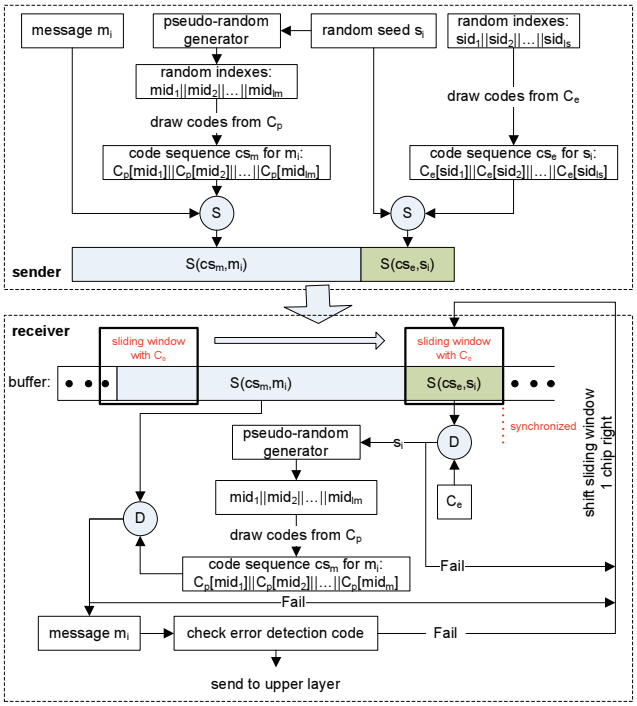


Figure 2: Delayed seed-disclosure DSSS (DSD-DSSS)

analysis of the signal.

4. BASIC DSD-DSSS

The basic idea of DSD-DSSS is two-fold. First, the code sequence used to spread a broadcast message is randomly generated based on a random seed only known to the sender. Thus, nobody except for the sender knows the right spreading code sequence before the sender discloses it. Second, the sender discloses the random seed at the end of the broadcast message, after the main message body has been transmitted. A receiver buffers received signal (or more precisely, received chips); it can decode the random seed and regenerate the spreading code sequence accordingly to despread the buffered chips. A jammer may certainly attempt the same thing. However, when the jammer recovers the seed and the spreading code sequence, all reachable receivers have already received the message. It is too late for the jammer to do any damage. Figure 2 illustrates the sending and receiving processes in DSD-DSSS. In the following, we describe this new scheme in detail.

4.1 Spreading Code Sets

Similar to traditional DSSS communication, DSD-DSSS uses spreading codes with good auto-correlation and low cross-correlation properties (e.g., PN codes).

DSD-DSSS keeps two sets of *publicly known* spreading codes: C_p and C_e. Codes in C_p are used to spread the message body m_i, while codes in C_e are used to spread the random seed at the end of each message. We require that C_p and C_e have no overlap (i.e., $C_p \cap C_e = \emptyset$). For convenience, we give each code in C_p (or C_e) a unique index. For a given index i for C_p (or C_e), we use $C_p[i]$ (or $C_e[i]$) to refer to the i-th code in C_p (or C_e).

We use individual bits in the message as the basic units of spreading. That is, each bit is spread with a different spreading code. As a result, even if an intelligent jammer can infer the spreading code for the current bit through real-time analysis, she cannot use this code to jam the following bit.

4.2 Sender

Given a l_m-bit message m_i, the sender encodes m_i in two parts: *message body* and *random seed*.

Spreading Message Body: The sender first generates a random seed s_i, and then uses a pseudo-random generator with seed s_i to generate a sequence of l_m random indexes $mid_1 \| mid_2 \| ... \| mid_{l_m}$, where $1 \leq mid_i \leq |C_p|$. The sender then generates a sequence of spreading codes cs_m for m_i by drawing codes from C_p using these indexes. That is, $cs_m = C_p[mid_1] \| C_p[mid_2] \| ... \| C_p[mid_{l_m}]$. The sender then uses cs_m to spread m_i (i.e., each code $C_p[mid_k]$ is used to spread the k-th bit of m_i). For convenience, we denote the spread message body (more precisely, the spread chips) as $S(cs_m, m_i)$.

Spreading Seed: A naive method is to disclose the seed s_i right after the spread message body $S(cs_m, m_i)$ so that receivers can recover s_i from the end of the message, generate cs_m using s_i, and despread the message. However, such a method is highly vulnerable to jamming attacks. Indeed, a jammer can simply disrupt the seed transmission to prevent the message from being received.

To prevent jamming attacks against the disclosed seed, the sender spreads the seed s_i using codes randomly selected from C_e, one of the public code sets. Assume the seed has l_s bits. The sender randomly draws l_s codes independently from C_e to form a sequence of l_s spreading codes, denoted $cs_s = C_e[sid_1] \| ... \| C_e[sid_{l_s}]$, where $sid_1, ..., sid_{l_s}$ are random integers between 1 and l_s. The sender then spreads the k-th bit in the seed s_i with the corresponding code $C_e[sid_k]$, where $1 \leq k \leq l_s$. The spreading results are then modulated, up-converted to the carrier frequency, and transmitted in the communication channel.

4.3 Receiver

As shown in Figure 2, each receiver keeps sampling the channel through down-conversion and demodulation, and saves the received chips in a cyclic buffer. Each receiver continuously processes the buffered chips to recover possibly received messages. To recover a meaningful message, a receiver has to first synchronize the buffered chips (i.e., align the buffered chips with appropriate spreading code) and then despread them.

Synchronization and Recovery of Seed: The goal of synchronization is to identify the positions of the chips of a complete message in the buffer before despreading them. The key for synchronization is to locate the seed, which occupies the last $l \times l_s$ chips in a message.

As shown in Figure 2, a receiver uses a sliding window with window size $l_s \times l$ to scan and locate the seed in the buffer, where l_s is the number of bits in a seed and l is the number of chips in a spreading code. The sliding window is shifted to the right by 1 chip each time.

In each scan, the receiver first uses the public code set C_e to despread the chips in the sliding window to synchronize with the sender. Conceptually, the receiver partitions the $l_s \times l$ chips into l_s groups, and tries each code in C_e to despread each group in the window. Note that using a set of codes with good auto-correlation and low cross-correlation properties, we can get high correlation and despread a bit successfully only when the same code (as the one used for spreading) is used to despread the encoded chips in the right position. If the despreading is successful for every group, the content in the window is a seed, which has been successfully recovered. At the same time, the position of the message body in the buffer is determined, i.e., the $l_m \times l$ chips to the left of the window in the buffer belong to the message body. Otherwise, the receiver shifts the window to the right by 1 chip and repeats the same process. This process can be further optimized. We omit the

details, since it is not critical for the presentation of our approach.

Despreading Message Body: Once a receiver recovers a seed s_i and determines the position of a received message in the buffer, it follows the same procedure as the sender to generate the sequence of spreading codes $cs_m = C_p[mid_1]||C_p[mid_2]||...||C_p[mid_{l_m}]$. The receiver then despreads the message body using cs_m. Specifically, the receiver partitions the chips buffered for the message body into l_m groups, each of which has l chips, and uses code $C_p[mid_k]$ to despread the k-th group of chips ($1 \leq k \leq l_m$).

At the end of this process, the receiver will recover the message body m_i and forward it to upper-layer protocols for further processing (e.g., error detection, signature verification).

4.4 Security Analysis

To show the effectiveness of DSD-DSSS against jamming attacks, we analyze the jamming probability in DSD-DSSS under different jamming attacks. Following the classification in [13], we consider two kinds of jamming attacks: *non-reactive* jamming and *reactive* jamming attacks. A non-reactive jammer continuously jams the communication channel without knowledge about actual transmissions, while a reactive jammer detects the transmission before jamming the channel. The jammer can apply three strategies to each attack: *static*, *sweep*, and *random* strategies. In the static strategy, the jammer uses the same code to jam the channel all the time. In the sweep strategy, the jammer periodically changes the code for jamming and does not reuse a code until all other codes have been used. In the random strategy, the jammer periodically changes the jamming code to a random code.

We also consider Denial of Service (DoS) attacks targeting at seed disclosure at receivers, in which the jammer attempts to force receivers to deal with a large number of candidate seeds.

4.4.1 Jamming Attacks

DSD-DSSS provides strong resistance against jamming attacks. Because each message is spread with a pseudo-random code sequence decided by a random seed, no one except for the sender can predict the spreading code sequence and jam the communication. The random seed is disclosed at the end of each message. Thus, when a jammer learns the seed, it is already too late to jam the transmitted message with it. A jammer may certainly try to jam the transmission of the random seed. However, each bit of the seed is spread with a code randomly selected from a code set (i.e., C_e), making it hard for a jammer to predict.

In the following, we provide a quantitative analysis of the jamming probabilities in various jamming scenarios. A jammer has two targets in each message: message body and seed. The jammer may jam the message body directly, or the seed so that receivers cannot recover the seed and then the spreading code sequence for the message body. To successfully jam even one bit of the message body, the jammer has to know the spreading code for that bit and synchronize her chips with those of the transmitted message.

Non-reactive Jamming Attacks: Non-reactive jammers do not rely on any information about the transmitted messages. Thus, they have to guess the spreading code and synchronization. We consider all three jamming strategies (i.e., static, sweep, and random) [13] and provide the jamming probabilities in the following two Theorems. The proofs are trivial and omitted due to space limit.

THEOREM 1. *When DSD-DSSS is used, the jamming probability of a non-reactive jammer with the static strategy is at most* $1 - \left(1 - \frac{1}{l|C_p|}\right)^{l_m}$ *if the jammer targets the message body, and is at most* $1 - \left(1 - \frac{1}{l|C_e|}\right)^{l_s}$ *if the jammer targets the seed.*

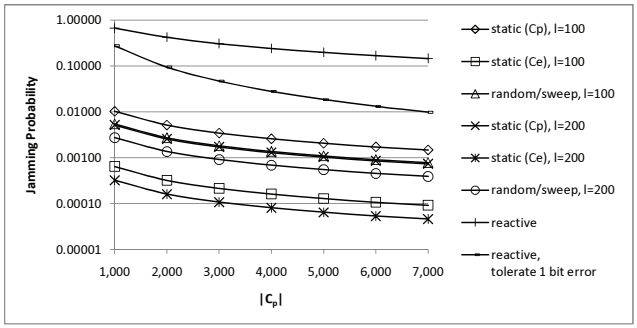


Figure 3: Maximum jamming probability for non-reactive and reactive jamming attacks ($l_m = 1024; l_s = 64; |C_e| = |C_p|; l = 100$ *or* 200)

THEOREM 2. *When DSD-DSSS is used, the jamming probability of a non-reactive jammer with the random (or sweep) strategy is at most* $1 - \left(1 - \frac{1}{l(|C_p|+|C_e|)}\right)^{l_m+l_s}$.

Reactive Jamming Attacks: A reactive jammer can detect the sender's transmission and perform real-time analysis of the transmitted signal. It can further synchronize with the sender so that she knows the precise chip layout of the transmitted message. However, as mentioned in Section 3, if a reactive jammer does not know the spreading code for any given bit data, she cannot jam the transmission based on real-time analysis. Nevertheless, the reactive jammer only needs to guess the sender's spreading code to jam the communication. This increases the jamming probability compared with simple non-reactive jamming attacks. Similar to non-reactive jammer, the reactive jammer can also use static, random, or sweep jamming strategies to jam the channel. We give the jamming probability for all three strategies in Theorem 3 below. (The proof is omitted due to space limit.) Note that the jamming strategy no longer has direct impact on the maximum jamming probability.

THEOREM 3. *When DSD-DSSS is used, the jamming probability of reactive jamming attacks is at most* $1 - \left(1 - \frac{1}{|C_p|}\right)^{l_m} \cdot \left(1 - \frac{1}{|C_e|}\right)^{l_s}$.

Figure 3 shows the jamming probabilities of both non-reactive and reactive jamming attacks, in which $|C_p| = |C_e|$, both ranging from 1,000 to 7,000, the sizes of message body and random seed are $l_m = 1,024$ bits and $l_s = 64$ bits, respectively, and the length l of each code is set to 100 or 200. Figure 3 shows that the reactive jamming attacks have much more impact than non-reactive jamming attacks due to the jammer's ability to synchronize with the sender. In all non-reactive jamming attacks, the jamming probabilities are no more than 0.01. However, even when $|C_p| = |C_e| = 7,000$, the reactive jammer's jamming probability is still 0.14. Figure 3 also shows that using Error Correction Code (ECC) can reduce the jamming probability dramatically. Simply using an ECC that can tolerate 1 bit error can lower the reactive jammer's jamming probability from 0.14 to 0.009.

The above results demonstrate that DSD-DSSS is effective in defending against jamming attacks, even when the jammer launches sophisticated reactive jamming attacks.

4.4.2 DoS Attacks against Seed Disclosure

DSD-DSSS has good resistance against various jamming attacks. However, an attacker may also inject bogus seeds or bogus messages, faking message transmissions from the sender. Indeed, this

is a problem common to all wireless communication systems. As long as a communication channel is accessible to an attacker, she can always inject fake messages. An authentication mechanism (e.g., digital signature) is necessary to filter out such fake messages.

An attacker may go one step further to launch DoS attacks targeting the seed disclosed at the end of each message. Specifically, the attacker may inject bogus seeds by continuously drawing a code from C_e, spreading a random bit, and transmitting it to receivers. A receiver will see a continuous stream of possible seeds being disclosed. Without any further protection, the receiver will have to attempt the decoding of a message with all possible seeds. An attacker may use multiple transmitters to inject multiple transmissions of each bit in a seed. As a result, the receiver may have to try the combinations of these options when decoding the messages. In Section 5, we will present an enhanced scheme to better protect seed disclosure against such DoS attacks in DSD-DSSS.

4.5 Performance Overheads

Computation Overhead and Delay: In terms of computation, the sender needs to generate a random seed, generate a spreading code sequence using a pseudo-random generator, and spread both the seed and the message body. All these operations can be performed efficiently and lead to negligible delay.

A receiver needs to synchronize with the sender's chips, despread and decode the seed, regenerate the spreading code sequence for the message body, and despread the message body. With the exception of synchronization and recovery of the seed, all other operations can be efficiently performed. Synchronization and recovery of seed are computationally expensive. A receiver should use all codes in C_e to despread every l chips in the buffer. Compared with traditional DSSS, this process is at least $|C_e|$ times more expensive.

DSD-DSSS introduces more receiver side delay than traditional DSSS, particularly because a receiver cannot start decoding a received message until the seed is recovered. Assume a straightforward implementation on the receiver side. For a received message, the time delay for the receiver to find the seed is $l(l_m+1)|C_e|t$, and the time delay to further recover the seed is $(l_s-1)|C_e|t$, where t is the time required to despread l chips. The sum of these two delays constitute the majority of the receiver side delay. Note that this process can be parallelized to reduce the receiver side delay.

Storage Overhead: DSD-DSSS requires a buffer to store the chips of a potential incoming message. When a message is being processed, a receiver has to buffer another message potentially being transmitted. Moreover, when there are multiple senders broadcasting at the same time, a receiver needs to buffer for decoded messages from all of them. Thus, in DSD-DSSS, a receiver needs storage that is possibly tens of times of that required by traditional DSSS. Nevertheless, considering the typical message size (e.g., a few hundred bytes) and the low cost of memory today, such a storage overhead is certainly affordable on a communication device.

Communication Overhead: DSD-DSSS adds a random seed at the end of each broadcast message, resulting in more communication overhead than traditional DSSS. Nevertheless, compared with the size of a typical message body (e.g., a few hundred bytes), the size of a random seed (e.g., 8 bytes) is negligible. Thus, DSD-DSSS introduces very light communication overhead.

5. EFFICIENT AND JAMMING-RESISTANT SEED DISCLOSURE

In this section, we enhance the basic DSD-DSSS scheme by developing a more effective protection of seed disclosure for the DoS threat discussed in Section 4.4.2. This approach gives normal receivers more advantages over jammers. It is based on the observation that a normal receiver can wait until a message is fully received to decode its content, while a jammer, to be effective in jamming, has to determine the jamming code when the message is being transmitted.

We propose *content-based code subset selection* for spreading and despreading the seed. The basic idea is to use the content of the seed to give some advantage to normal receivers. Specifically, the sender spreads the seed bit-by-bit from the end to the beginning. For each bit (except for the last one), the sender uses both the value and the spreading code of the later bit to determine its candidate spreading codes, which are a small subset of all possible codes. Note that when a receiver starts decoding a message, it already has the entire message buffered. Thus, a receiver can follow the same procedure as the sender to recover the small subset of candidate codes for each bit of the seed. However, without the complete message, a jammer has to consider many more spreading codes. Any code not in the right subset will be ignored by normal receivers. Moreover, even if some codes chosen by jammers are accepted by chance, the receivers do not need to consider the combinations of all accepted codes in different bit positions in the seed, avoiding the most serious DoS attack.

The basic DSD-DSSS scheme employs two public code sets C_p and C_e, where only C_e is used to spread the seed. In the new approach, we enhance the protection of the seed by using both code sets. The codes in C_e are only used to spread the last bit of the seed, marking the end of the seed. We generate multiple subsets of C_p. Each earlier bit of the seed is spread with one of these subsets, selected based on the value and spreading code of the later bit.

A reactive jammer may attempt to infer the code used to spread the next bit based on her current observation (i.e., the code used for the current bit). It is critical not to give the jammer such an opportunity. Thus, we require that each code appear in multiple subsets of C_p. As a result, knowing the code for the current and past bits does not give any jammer enough information to make inference for future bits.

5.1 Generation of Subsets of C_p

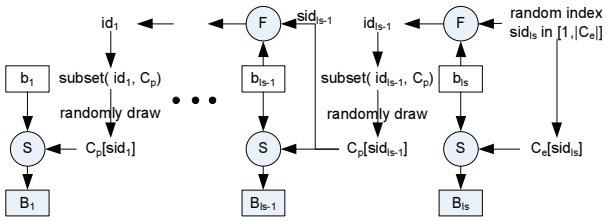
To meet the requirement for the subsets of C_p, as a convenient starting point, we choose *finite projective plane*, which is a symmetric Balanced Incomplete Block Design (BIBD) [8], to organize the spreading codes in C_p. It is certainly possible to use other combinatorial design methods to get better properties. We consider these as possible future work, but do not investigate them in this paper.

A *finite projective plane* has n^2+n+1 points, where n is an integer called the *order* of the projective plane [8]. It has n^2+n+1 lines, with $n+1$ points on every line, $n+1$ lines passing through every point, and every two points appearing together on exactly 1 line. It is shown in [8] that when n is a power of a prime number, there always exists a finite projective plane of order n.

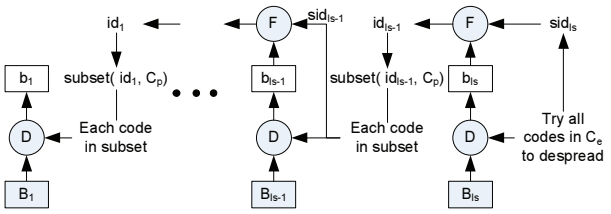
In this paper, we consider the points on a finite projective plane as spreading codes in C_p and lines as subsets of C_p. For a finite projective plane with order n, we associate each point with a spreading code and each line with a subset. We construct C_p by selecting n^2+n+1 spreading codes with good auto-correlation and low cross-correlation properties (e.g., PN codes [6]). As a result, we also have n^2+n+1 subsets, where each subset has $n+1$ codes, each code appears in $n+1$ subsets, and every two codes co-exist in exactly 1 subsets. We give a unique index to each subset of C_p to facilitate the selection of subsets during spreading and despreading.

5.2 Spreading the Seed

Figure 4(a) shows how the sender spreads the seed. We represent



(a) Spreading the seed



(b) Despreading the seed

Figure 4: Content-based code subset selection

each bit of the seed as b_i, where $1 \leq i \leq l_s$ and l_s is the number of bits in the seed. As mentioned earlier, the sender spreads the seed from the end to the beginning.

For bit b_{l_s}, the sender randomly chooses a code from C_e and spreads b_{l_s} with this code to get a sequence of chips B_{l_s}. Assume the index of the chosen code is sid_{l_s}, where $1 \leq sid_{l_s} \leq |C_e|$.

We use a function F to determine which subset of C_p is used for the next (earlier) bit. Function F has two inputs: the index of a code in C_p or C_e, and a bit value (1 or 0). The output of F is the index of a subset of C_p. F can be any function that reaches the indexes of the subsets of C_p evenly with evenly distributed inputs. To guarantee that any subset of C_p be used for b_{l_s-1}, we must have $|C_e| \geq \left\lceil \frac{|C_p|}{2} \right\rceil$. For simplicity, we set $|C_e| = \left\lceil \frac{|C_p|}{2} \right\rceil$. Specifically, for bit b_i, where $1 \leq i \leq l_s - 1$, the sender uses sid_{i+1} and b_{i+1} as the input of F to get id_i, the index of subset for bit b_i. The sender then randomly draws a code from the subset of C_p with index id_i to spread bit b_i and get the sequence of chips B_i. Assume that the code's index is sid_i. The sender continues this process to spread the earlier bits.

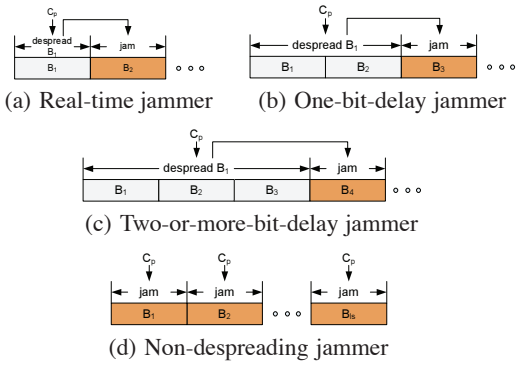
5.3 Despreading the Seed

Figure 4(b) shows how a receiver despreads the seed. The receiver continuously tries to find the end of a message in the buffer using a sliding window method as discussed in Section 4.

In the sliding window, the receiver sequentially tries every code in C_e to despread the last l chips in the window. If no code in C_e can successfully despread the last l chips, the sliding window shifts 1 chip to the right in the buffer. If the code with index sid_{l_s} can successfully despread the last l chips to get a bit value b_{l_s}, the sliding window potentially covers a seed.

The receiver despreads the seed bit-by-bit from the end to the beginning. After getting b_{l_s}, the receiver uses sid_{l_s} and b_{l_s} as the input to function F to get id_{l_s-1}, the index of the subset of C_p used for bit b_{l_s-1}. The receiver then sequentially tries each code in this subset to despread the l chips for bit b_{l_s-1}, until it finds the correct code. Assume the index of this code is sid_{l_s-1} and the decoded bit value is b_{l_s-1}. The sender then repeats this process to decode the earlier bits b_{l_s-2}, ..., b_1, and eventually reconstructs the seed $b_1||b_2||...||b_{l_s}$.

During this process, if any despreading failure occurs, the receiver gives up the current decoding process and shifts the sliding window by 1 chip to the right to look for the next seed candidate. Once the receiver gets the seed $b_1||b_2||...||b_{l_s}$, it uses this seed to generate the spreading code sequence for the message body and despreads the message body as discussed in Section 4.



(a) Real-time jammer (b) One-bit-delay jammer

(c) Two-or-more-bit-delay jammer

(d) Non-despreading jammer

Figure 5: Reactive jamming with different capabilities

5.4 Analysis

The objective of our analysis is to understand (1) the effectiveness of content-based code subset selection in enhancing DSD-DSSS's anti-jamming capability, and (2) the capability of this mechanism against DoS attacks discussed in Section 4.4.

5.4.1 Effectiveness against Jamming Attacks

We analyze the probability of an attacker jamming the seed to show the effectiveness of content-based code subset selection. Moreover, this scheme also increases the difficulty for a jammer to identify the right spreading code compared with a normal receiver. We thus analyze the search space (i.e., the set of candidate spreading codes) for both a receiver and a jammer to demonstrate the advantage of a normal receiver over a jammer.

We consider jammers with four levels of computation capabilities: (1) real-time, (2) one-bit-delay, (3) two-or-more-bit-delay, and (4) non-despreading jammers. All jammers are reactive jammers that can synchronize with the sender. The first three types of jammers perform despreading and online analysis to assist jamming, which improves the jamming probability by reducing the number of candidate spreading codes (i.e., possible codes used by the sender).

As illustrated in Figure 5(a), a real-time jammer has intensive computation power to finish the analysis and identify the spreading code used for bit 1 (represented by chips B_1), and can use this information to jam the immediately following bit (represented by chips B_2). As shown in Figures 5(b) and 5(c), a one-bit-delay jammer and a two-or-more-bit-delay jammer need additional time, equivalent to the time for transmitting 1 bit and 2 or more bits, respectively, to finish online analysis before applying the result for jamming purposes. Thus, after learning the spreading code for bit 1, a one-bit-delay jammer and a two-or-more-bit-delay jammer can only jam bit 3 (represented by chips B_3) and bit 4 (represented by chips B_4) or later, respectively. These jammers may certainly perform the same analysis of every bit they receive and use the analysis result to jam future bits. A non-despreading jammer simply skips the despreading step and use C_e to jam the last bit of the seed and use C_p to jam the remaining part of the seed, as Figure 5(d) shows.

In the following, we prove Lemma 1 to assist the analysis.

LEMMA 1. *Given k distinct subsets, the number of codes that can be used to derive these subsets by applying function F is in the range of $[k, \min\{2k, n^2 + n + 1\}]$.*

PROOF. Since the output of function F is evenly distributed

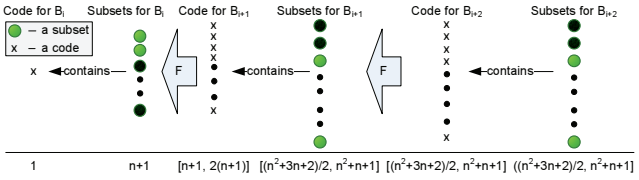


Figure 6: Jammers' view of spreading codes and subsets (Assume the jammer just derived code x for bit b_i (chips B_i))

Table 2: Search spaces

real time	$n^2 + (l_s - 1)n + (l_s - 1)$
1 bit delay	$2(n^2 + n + 1) + (l_s - 4)\frac{(n+1)(n+2)}{2}$
q bits delay ($q \geq 2$)	$(q+1)(n^2 + n + 1) + (l_s - 2(q+1))\frac{(n+1)(n+2)}{2}$

when the inputs are evenly distributed, for each subset, there are two possible codes as inputs. For each code, there are two possible subsets as outputs. Thus, the lower bound is k and the upper bound is $\min\{2k, n^2 + n + 1\}$. □

Real-time Jammers: If a jammer can despread each bit in real-time (e.g., by using parallel computing devices), the jammer can know the code for despreading B_i once the transmission of B_i is complete. As Figure 6 shows, the jammer can then identify all $n+1$ subsets that contain this code. By using the inverse of function F, the jammer can also identify all possible codes in C_p that were used to determine these subsets, which were also used to spread b_{i+1} into B_{i+1}. The number of possible codes for B_{i+1} is in the range of $[n+1, 2(n+1)]$, according to Lemma 1. Thus, the jammer can jam the transmission of B_{i+1} by randomly selecting a code from these codes (rather than from C_p). Since the last bit of the seed is spread using codes in C_e, the number of all possible codes for the jammer is thus in the range of $[n+1, \min\{2(n+1), |C_e|\}]$.

In the worst case, a real-time jammer can despread all bits of the seed except for B_{l_s} and jams all bits. The jamming probability of the first bit is at most $\frac{1}{|C_p|}$, the jamming probability of the last bit is at most $P_{e0} = \frac{1}{n+1}$, and the jamming probability of B_i ($2 \leq i \leq l_s - 1$) is at most $P_{p0} = P_{e0} = \frac{1}{n+1}$. Thus, the jamming probability of the seed is at most

$$P_{\text{real-time}} = 1 - \left(1 - \frac{1}{|C_p|}\right)(1 - P_{p0})^{l_s - 1}.$$

By including an ECC that can tolerate 1 bit error, we can reduce the maximum jamming probability to

$$P_{\text{real-time}} = 1 - (1 - P_{p0})^{l_s - 1} - (l_s - 1)\left(1 - \frac{1}{|C_p|}\right)P_{p0}(1 - P_{p0})^{l_s - 2}.$$

It is easy to see that the total search space for a real-time jammer throughout all bits of the seed is at least

$$SS_{\text{real-time}} = |C_p| + (l_s - 2)(n+1) = n^2 + (l_s - 1)n + (l_s - 1).$$

Non-real-time Jammers: The results for one-bit-delay, two-or-more-bit-delay, and non-despreading jammers can be derived similarly. Due to the space limit, we do not show the details but list the final results for the jamming probabilities and search spaces in Table 1 and Table 2, respectively.

Comparison of Jamming Probabilities: Figure 7 shows the maximum jamming probabilities of the four types of jammers against the random seed with reasonable parameters. Recall that the size of C_p is determined by parameter n (i.e., $C_p = n^2 + n + 1$). Thus, we use parameter n as the x-axis in this figure. To better see the impact of ECC, we also include the maximum jamming probabilities assuming an ECC is used in the seed to tolerate 1 bit error.

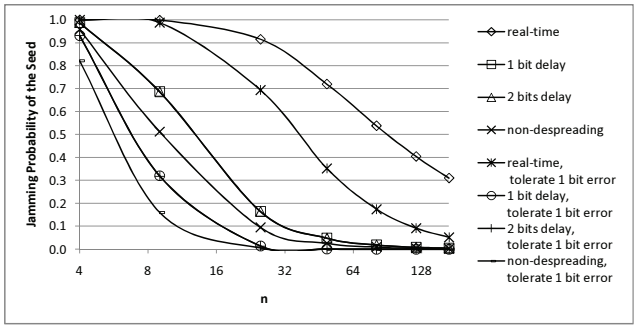


Figure 7: Maximum jamming probability against seed ($n = 4, 9, 25, 49, 81, 121, 169$; $l_s = 64$; $|C_e| = \left\lceil \frac{|C_p|}{2} \right\rceil$)

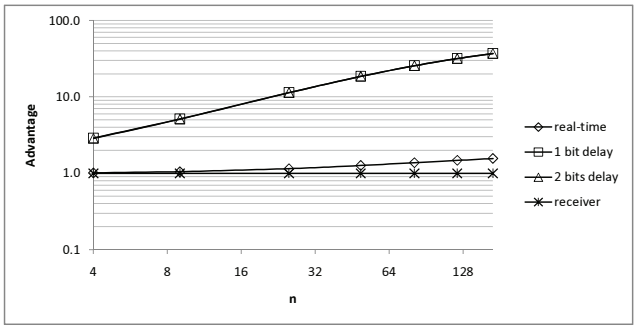


Figure 8: Advantage of receivers over jammers ($n = 4, 9, 25, 49, 81, 121, 169$; $l_s = 64$; $|C_e| = \left\lceil \frac{|C_p|}{2} \right\rceil$)

Figure 7 shows that the real-time jammer has the highest jamming probability among all jammers. However, we would like to point out that the real-time jammer is a strong assumption; such a jammer may have to use special hardware (e.g., parallel computing devices) to obtain the despreading results. As the jammer has to tolerate 1 or 2 bit delays, the maximum jamming probability decreases significantly. Not surprisingly, the non-despreading jammer has the lowest jamming probability.

Figure 7 also shows that increasing n (and thus $|C_p|$) can quickly reduce the maximum jamming probability for all types of jammers. Moreover, the application of ECC can also reduce the jamming probability effectively, though it introduces additional computational and communication overheads. For example, with an ECC tolerating just 1 bit error, we can reduce the real-time jammer's maximum jamming probability from 0.31 to 0.05 when $n = 169$. Further increasing n or the number of bit errors the ECC can tolerate can quickly reduce the maximum jamming probability to a negligible level.

Comparison of Search Spaces: Now let us compare the numbers of candidate spreading codes that a normal receiver and a reactive jammer have to consider, respectively. Such numbers represent the computational costs they have to spend. Since a receiver buffers the complete seed before despreading it, it can despread the last bit of the seed first to learn sid_{l_s}, and then infer the indexes of subsets for previous bits of the seed. The size of total search space for a receiver is thus $(l_s - 1)(n+1) + |C_e|$. To show the advantage of a receiver over a jammer, we compute function $Adg = \frac{SS_j}{SS_r}$ for real-time, one-bit-delay, two-or-more-bit-delay jammers, where SS_j and SS_r are the sizes of the total search space for the jammer and the receiver, respectively. The larger Adg is, the more advantage the receiver has over the jammer.

Table 1: Jamming probabilities for jammers with different jamming capabilities ($|C_p| = n^2+n+1$; $C_e = \left\lceil \frac{|C_p|}{2} \right\rceil$; $P_{p0} = \frac{1}{n+1}$; $P_{p1} = \frac{2}{(n+1)(n+2)}$; $P_{e1} = P_{e2} = \frac{1}{|C_e|}$; $P_{p2} > \frac{2}{(n+1)(n+2)}$)

real time	$1-\left(1-\frac{1}{\lvert C_p\rvert}\right)(1-\frac{1}{n+1})^{l_s-1}$
1 bit delay	$1-\left(1-\frac{1}{\lvert C_p\rvert}\right)^2(1-P_{p1})^{l_s-3}(1-P_{e1}))$
q bits delay ($q \geq 2$)	$1-\left(1-\frac{1}{\lvert C_p\rvert}\right)^{q+1}(1-P_{p2})^{l_s-q-2}(1-P_{e2})$
non-despreading	$1-\left(1-\frac{1}{\lvert C_p\rvert}\right)^{l_s-1}\left(1-\frac{1}{\lvert C_e\rvert}\right)$
real time, tolerate 1 bit error	$1-(1-P_{p0})^{l_s-1}-(l_s-1)\left(1-\frac{1}{\lvert C_p\rvert}\right)P_{p0}(1-P_{p0})^{l_s-2}$
1 bit delay, tolerate 1 bit error	$1-\left(1-\frac{1}{\lvert C_p\rvert}\right)^2(1-P_{p1})^{l_s-3}(1-P_{e1})-\frac{2}{\lvert C_p\rvert}\left(1-\frac{1}{\lvert C_p\rvert}\right)(1-P_{p1})^{l_s-3}(1-P_{e1})-(l_s-3)\left(1-\frac{1}{\lvert C_p\rvert}\right)^2P_{p1}(1-P_{p1})^{l_s-4}(1-P_{e1})$
q bits delay ($q \geq 2$), tolerate 1 bit error	$1-\left(1-\frac{1}{\lvert C_p\rvert}\right)^{q+1}(1-P_{p2})^{l_s-q-2}-(q+1)\frac{1}{\lvert C_p\rvert}\left(1-\frac{1}{\lvert C_p\rvert}\right)^q(1-P_{p2})^{l_s-q-2}(1-P_{e2})-(l_s-q-2)\left(1-\frac{1}{\lvert C_p\rvert}\right)^{q+1}P_{p2}(1-P_{p2})^{l_s-q-3}(1-P_{e2})$
non-despreading, tolerate 1 bit error	$1-\left(1-\frac{1}{\lvert C_p\rvert}\right)^{l_s-1}-(l_s-1)\frac{1}{C_p}\left(1-\frac{1}{\lvert C_p\rvert}\right)^{l_s-2}\left(1-\frac{1}{\lvert C_e\rvert}\right)$

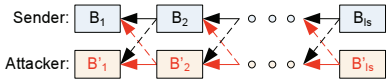


Figure 9: Seed recovery in presence of bogus seed transmission

Figure 8 shows the advantage of a receiver over the jammers. (The non-despreading jammer is not included, since she does not despread at all.) All jammers have larger search space than the receiver, and the gap grows wider when n increases. The real-time jammer remains the most powerful one; she can reduce the search space for the next bit dramatically by despreading the current bit, and thus has the smallest search space among all jammers, which is close to the receiver's search space. Nevertheless, Figure 8 considers the lower bound of the jammers' search space. Moreover, there is still observable difference between the search spaces of the real-time jammer and the receiver. The search spaces of the one-bit-delay and two-or-more-bit-delay jammers have almost the same size, which are significantly larger than that of the receiver.

5.4.2 *Effectiveness against DoS Attacks*

As discussed in Section 4.4, a jammer can transmit bogus seeds or even entire bogus messages. As long as the communication channel is available to attackers, they can always inject bogus messages. Thus, in general, this is an unavoidable problem in presence of compromised receivers. When these bogus seeds are not concurrently transmitted and do not overlap with the sender's normal seed transmission, a receiver can filter them out using error detection coding and broadcast authentication (e.g., digital signature). However, when the bogus seeds do overlap with the normal seed, the receiver will have to consider all combinations of options for each bit of the seed, thus suffering from serious DoS attacks.

The proposed content-based code subset selection scheme can effectively mitigate such situations by chaining the codes used to spread different bits of the seed. To demonstrate the effectiveness of this approach, we show the number of candidate seeds when the jammer synchronizes with a sender and transmits a bogus seed $(B'_1||B'_2||...||B'_{l_s})$ to interfere with the transmission of the actual seed $(B_1||B_2||...||B_{l_s})$, as shown in Figure 9.

Intuition: During seed recovery, a receiver will attempt to recover the seed starting with both B_{l_s} and B'_{l_s}. The number of seed candidates is the number of paths starting from B_{l_s} or B'_{l_s} and ending at B_1 or B'_1. In the basic DSD-DSSS, the receiver will try all possible paths shown in Figure 9. However, the content-based code subset selection scheme can constrain the paths between two seeds (dashed lines) during despreading. Intuitively, the jammer does not know which code subset is used to spread each bit of the seed at the time of her transmission, and thus cannot select the right code, which will be considered valid by a receiver during despreading. If the code for the i-th bit ($1 \leq i \leq l_s$) of the bogus seed is not in the subset for the i-th bit of the good seed, the receiver will not consider it for despreading the i-th bit of the bogus seed. As a result, the path from the good seed to the bogus one (in black dashed lines) will not exist. Similarly, if the code for i-bit of the good seed is not in the subset for i-th bit of the bogus seed, the receiver will not consider it for despreading the i-th bit of the good seed. Thus, the path from the bogus seed to the good one (in red dashed lines) will not exist.

During the analysis, we consider *non-despreading*, *real-time*, *one-or-more-bit-delay* jammers to see the best-case scenarios for the jammers when they can benefit from knowing a part of the seed and spreading codes. The capability of these jammers is the same as discussed earlier during the analysis of jamming probabilities. However, the objective of these jammers now is to trigger the receiver to have more seed candidates during despreading by injecting bogus seeds. We assume these jammers can perform despreading and transmitting operations at the same time, though they can only use the despreading results of each bit for later bits.

Non-despreading Jammers: If the jammer follows the sender's procedure to send the seed, the probability of having a path from from B'_{i+1} to B_i (red dashed line) and the probability of having a path from from B_{i+1} to B'_i (black dashed line) are both $\frac{1}{n^2+n+1}$, because any pair of codes only exist in exactly one subset. Only one among the n^2+n+1 subsets can despread the i-th bit of both the bogus and the good seeds. The expected number of seed candidates is thus $2(1+\frac{1}{|C_e|})(1+\frac{1}{|C_p|})^{l_s-2}$ according to Theorem 4. The proof of Theorem 4 is omitted due to the space limit.

THEOREM 4. *When there is a non-despreading jammer launching the DoS attack against seed disclosure, the expected number of seed candidates is* $2(1+p_1)(1+p_2)^{l_s-2}$. *Among them,* $(1+p_1)(1+p_2)^{l_s-2}$ *paths end at* B_1, *and* $(1+p_1)(1+p_2)^{l_s-2}$ *paths end at* B'_1, *where* $p_1 = \frac{1}{|C_e|}$ *and* $p_2 = \frac{1}{|C_p|}$.

Real-time and one-or-more-bit-delay Jammers: Similar to the analysis for non-despreading jammer, we analyze the expected number of seed candidates caused by real-time and one-or-more-bit-delay jammers. Due to the space limit, we simply list results and omit proofs. The expected number of seed candidates caused by real-time jammer is smaller than $2(1+\frac{1}{|C_p|})(1+\frac{n}{(n+1)^2})^{l_s-2}$,

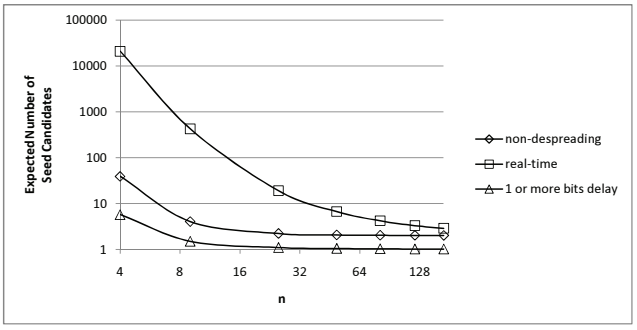


Figure 10: Expected number of seed candidates for normal receiver under DoS attacks against seed disclosure ($n = 4, 9, 25, 49, 81, 121, 169$; $l_s = 64$; $|C_e| = \left\lceil \frac{|C_p|}{2} \right\rceil$)

and that caused by one-or-more-bit-delay jammer is smaller than $(1+2p_2)(1+p_2)E_3 + (p_4+2p_2)(1+p_2)E_3'$, where

$E_3 = 1 + \frac{p_4}{\lambda_1-\lambda_2} \cdot \frac{(2p_5-\lambda_2)(1-\lambda_1^{l_s-3})}{1-\lambda_1} + \frac{p_4}{\lambda_1-\lambda_2} \cdot \frac{(\lambda_1-2p_5)(1-\lambda_2^{l_s-3})}{1-\lambda_2}$,

$E_3' = \frac{2p_5-\lambda_2}{\lambda_1-\lambda_2} \cdot \lambda_1^{l_s-3} + \frac{\lambda_1-2p_5}{\lambda_1-\lambda_2} \cdot \lambda_2^{l_s-3}$, $p_2 = \frac{1}{|C_p|}$, $p_4 = \frac{2}{n+2}$,

$p_5 = \frac{2n(n+3)}{(n+1)^2(n+2)^2}$, $\lambda_1, \lambda_2 = \frac{1+p_5 \pm \sqrt{(1+p_5)^2-4(1-p_4)p_5}}{2}$.

Comparison: Figure 10 shows the expected numbers of seed candidates caused by non-despreading, real-time, and one-bit-delay jammers when they launch DoS attacks against seed disclosure. The more seed candidates the receiver has, the more computational cost the receiver has to spend receiving a message. Among three of them, the real-time jammer has the highest impact. However, it is still limited when n is reasonably large. The number of seed candidates is below 10 for all jammers when $n \geq 49$. The non-despreading jammer and the one-bit-delay jammers do not introduce much overhead to the receiver. The expected number of seed candidates by the non-despreading jammer is below 4 when $n \geq 9$. The expected number of seed candidates by the one-bit-delay jammer is below 1.5 when $n \geq 9$. When $n = 169$, the expected number of seed candidates of non-despreading, real-time, and one-bit-delay jammers are only 2, 2.87, and 1.01, respectively. Note that the lines shown in Figure 10 are conservative estimates showing the upper bound of the expected impact these jammers can introduce.

Compared with the basic DSD-DSSS scheme, in which the jammer can introduce 2^{l_s} seed candidates (e.g., 2^{64} seed candidates using the same parameters in Figure 10), the content-based code subset selection scheme has significantly reduced the impact of the DoS attacks against seed disclosure. Thus, it provides effective defense against such DoS attacks.

6. EXPERIMENTAL EVALUATION

We have implemented a prototype of DSD-DSSS based on GNU Radio [1] using Universal Software Radio Peripherals (USRPs) with XCVR2450 daughter boards [12]. Our implementation includes both the basic DSD-DSSS scheme (named DSD-DSSS BASIC) and the enhanced DSD-DSSS with content-based code subset selection (named DSD-DSSS SUBSET). We have also implemented DSSS [6] and UDSSS [15] as references in our experimental evaluation.

In our experiments, we used two USRPs with XCVR2450 daughter boards, one as the sender, and the other as the receiver. The sender was connected to a laptop (Intel Core 2 Duo @ 2.6GHz), while the receiver was connected to a desktop PC (Intel Pentium 4 @ 3.2GHz), both through 480 Mbps USB 2.0 links. Both the laptop and the desktop ran Ubuntu 9.04 and GnuRadio 3.2. The payload size in spreading/despreading module was configured to be 256, 512, or 1024 bits. We measured the receiver's average despreading time of a message for 200 rounds. Since messages were sent consecutively, the despreading of all messages after the first message was automatically synchronized (i.e., knowing the starting chip of each message). For DSD-DSSS, we set the seed size as 64 bits and used SAS v9.1.3 [17] to generate BIBD subsets of C_p. We used SHA-1 to as the pseudo-random number generator for both DSD-DSSS and UDSSS schemes.

Figure 11(a) shows the average despreading time of a message for DSD-DSSS BASIC, DSD-DSSS SUBSET, UDSSS, and DSSS schemes when using different size of code set. For DSD-DSSS, $|C_p| = n^2 + n + 1$, $|C_e| = \left\lceil \frac{|C_p|}{2} \right\rceil$, where $n \in [2, 20]$. For UDSSS, the number of code sequences is the same as the number of codes in $|C_p|$. As Figure 11(a) shows, DSSS is the most efficient scheme because only one code sequence is used to despread messages. UDSSS is slower than DSSS since it has to check the first code of all code sequences.

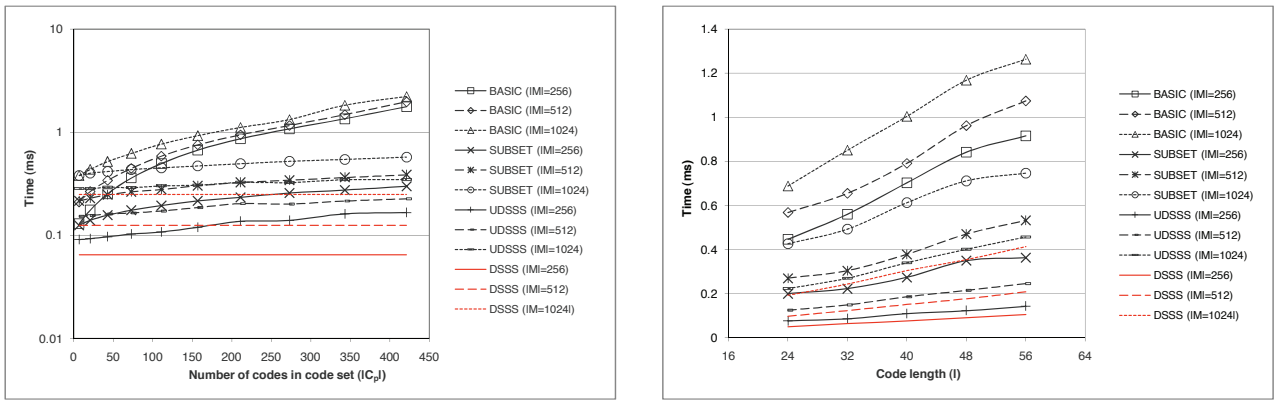
UDSSS is more efficient than DSD-DSSS because DSD-DSSS has to check $64 \cdot |C_e| = 64 \cdot \left\lceil \frac{|n^2+n+1}{2} \right\rceil$ codes for BASIC scheme and $63 \cdot (n+1) + |C_e| = 63 \cdot (n+1) + \left\lceil \frac{n^2+n+1}{2} \right\rceil$ codes for SUBSET scheme, while UDSSS only needs to check $|C_p| = n^2 + n + 1$ codes. DSD-DSSS BASIC always has the largest number of codes to check. DSD-DSSS SUBSET scheme has larger number of codes to check than UDSSS when $n < 126$ (i.e., $|C_p| < 16003$). When $n \geq 126$, DSD-DSSS SUBSET scheme would be even more efficient than UDSSS. However, we cannot run the evaluation for $n \geq 126$ due to the large computational power requirement.

Figure 11(b) shows the average despreading time of a message for different code lengths ($l = 24, 32, 40, 48, 56$). It is obvious that all DSD-DSSS, UDSSS, and DSSS need more time to despread messages when the code length is increased. The despreading time of DSD-DSSS BASIC increases much faster than that of other schemes due to the much larger search space of codes. DSSS is still the most efficient scheme, and UDSSS is more efficient than DSD-DSSS. Although UDSSS is faster than DSD-DSSS in both Figure 11(a) and Figure 11(b), UDSSS suffers from the reactive jamming attack [15] while DSD-DSSS does not.

7. RELATED WORK

Spread spectrum wireless communication techniques, including DSSS and FH, have been commonly used for anti-jamming communication [6]. However, as discussed earlier, traditional spread spectrum techniques all require pre-shared secret keys, and are not suitable for broadcast communication where there may be compromised or malicious receivers. We have discussed most closely related works in the introduction, including UFH and its variations [7, 18–20], UDSSS [14, 15], and BBC [2, 3]. We do not repeat them here. An idea similar to ours was also proposed in [7]; however, it is targeted at spread spectrum based pairwise communication, and does not provide the protection of seed as in our scheme. RD-DSSS provides the anti-jamming capability by encoding each bit of data using the correlation of unpredictable spreading codes [11].

There are other related work, including approaches for detecting jamming attacks [23], identifying insider jammers [4,5], mitigating jamming of control channels [9, 21], jamming avoidance and evasion [2,22,24], and mitigating jamming in sensor networks [10,22]. Our technique is complementary to these techniques.



(a) for different code set sizes ($l = 32$)

(b) for different code lengths ($|C_p| = 111$)

Figure 11: Comparison of time to despread message in DSSS, UDSSS, and DSD-DSSS

8. CONCLUSION

In this paper, we proposed DSD-DSSS, an efficient anti-jamming broadcast communication scheme. It achieves anti-jamming capability through randomly generating the spreading code sequence for a broadcast message through a random seed and delaying the disclosure of the seed at the end of the message. We also developed an effective protection for the disclosure of the random seed through content-based code subset selection. Our analysis in this paper demonstrated that this suite of techniques can effectively defeat jamming attacks. Our implementation and evaluation shows the feasibility of DSD-DSSS in real world. We measured the performance of DSD-DSSS without jamming attacks due to the time limitation. Although DSD-DSSS is slower than UDSSS without jamming attacks, DSD-DSSS may be faster than UDSSS in presence of jammers. We will verify this in our future work.

9. REFERENCES

[1] GNU Radio - The GNU Software Radio. `http://www.gnu.org/software/gnuradio/`.

[2] L. Baird, W. Bahn, and M. Collins. Jam-resistant communication without shared secrets through the use of concurrent codes. Technical report, US Air Force Academy, 2007.

[3] L. C. Baird, W. L. Bahn, M. D. Collins, M. C. Carlisle, and S. C. Butler. Keyless jam resistance. In *Proceedings of the IEEE Information Assurance and Security Workshop*, pages 143–150, June 2007.

[4] J. Chiang and Y. Hu. Extended abstract: Cross-layer jamming detection and mitigation in wireless broadcast networks. In *Proceedings of ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc '07)*, 2007.

[5] J. Chiang and Y. Hu. Dynamic jamming mitigation for wireless boradcast networks. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM '08)*, 2008.

[6] A. Goldsmith. *Wireless Communications*. Cambridge University Press, August 2005.

[7] T. Jin, G. Noubir, and B. Thapa. Zero pre-shared secret key establishment in the presence of jammers. In *Procceedings of MobiHoc '09*, May 2009.

[8] D. L. Kreher and D. Stinson. *Combinatorial Algorithms: Generation, Enumeration, and Search*. CRC Press, 1999.

[9] L. Lazos, S. Liu, and M. Krunz. Mitigating control-channel jamming attacks in multi-channel ad hoc networks. In *Proceedings of 2nd ACM Conference on Wireless Networking Security (WiSec '09)*, March 2009.

[10] M. Li, I. Koutsopoulos, and R. Poovendran. Optimal jamming attacks and network defense policies in wireless sensor networks. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM '07)*, 2007.

[11] Y. Liu, P. Ning, H. Dai, and A. Liu. Randomized differential dsss: Jamming-resistant wireless broadcast communication. In *Proceedings of the 2010 IEEE INFOCOM*, 2010.

[12] Ettus Research LLC. The USRP product family products and daughter boards. `http://www.ettus.com/products`. Accessed in August 2010.

[13] R. Poisel. *Modern Communications Jamming Principles and Techniques*. Artech House Publishers, 2006.

[14] Pöpper, M. Strasser, and S. Čapkun. Anti-jamming broadcast communication using uncoordinated spread spectrum techniques. *IEEE Journal on Selected Areas in Communications: Special Issue on Mission Critical Networking*, 2010.

[15] C. Pöpper, M. Strasser, and S. Čapkun. Jamming-resistant broadcast communication without shared keys. In *Proceedings of the USENIX Security Symposium*, 2009.

[16] J. Proakis. *Digital Communications*. McGraw-Hill, August 2000.

[17] SAS. Business analytics and business intelligence software. `http://www.sas.com`.

[18] D. Slater, P. Tague, R. Poovendran, and B. Matt. A coding-theoretic approach for efficient message verification over insecure channels. In *Procceedings of the 2nd ACM Conference on Wireless Networking Security (WiSec '09)*, pages 151–160, March 2009.

[19] M. Strasser, C. Pöper, S. Čapkun, and M. Čagalj. Jamming-resistant key establishment using uncoordinated frequency hopping. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 64–78, 2008.

[20] M. Strasser, C. Pöpper, and S. Čapkun. Efficient uncoordinated FHSS anti-jamming communication. In *Procceedings of MobiHoc '09*, May 2009.

[21] P. Tague, M. Li, and R. Poovendran. Probabilistic mitigation of control channel jamming via random key distribution. In *Proceedings of IEEE 18th International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC '07)*, pages 1–5, 2007.

[22] W. Xu, W. Trappe, and Y. Zhang. Channel surfing: Defending wireless sensor networks from jamming and interference. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks (IPSN '07)*, 2007.

[23] W. Xu, W. Trappe, Y. Zhang, and T. Wood. The feasibility of launching and detecting jamming attacks in wireless networks. In *Proceedings of ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc '05)*, 2005.

[24] W. Xu, T. Wood, W. Trappe, and Y. Zhang. Channel surfing and spatial retreats: Defenses against wireless denial of service. In *Proceedings of the 3rd ACM Workshop on Wireless Security (WiSe '04)*, 2004.

Always Up-to-date – Scalable Offline Patching of VM Images in a Compute Cloud

Wu Zhou Peng Ning
North Carolina State University
{wzhou2, pning}@ncsu.edu

Xiaolan Zhang Glenn Ammons
IBM T.J. Watson Research Center
{cxzhang, ammons}@us.ibm.com

Ruowen Wang
North Carolina State University
rwang9@ncsu.edu

Vasanth Bala
IBM T.J. Watson Research Center
vbala@us.ibm.com

ABSTRACT

Patching is a critical security service that keeps computer systems up to date and defends against security threats. Existing patching systems all require running systems. With the increasing adoption of virtualization and cloud computing services, there is a growing number of dormant virtual machine (VM) images. Such VM images cannot benefit from existing patching systems, and thus are often left vulnerable to emerging security threats. It is possible to bring VM images online, apply patches, and capture the VMs back to dormant images. However, such approaches suffer from unpredictability, performance challenges, and high operational costs, particularly in large-scale compute clouds where there could be thousands of dormant VM images.

This paper presents a novel tool named *Nüwa* that enables efficient and scalable offline patching of dormant VM images. Nüwa analyzes patches and, when possible, converts them into patches that can be applied offline by rewriting the patching scripts. Nüwa also leverages the VM image manipulation technologies offered by the Mirage image library to provide an efficient and scalable way to patch VM images in batch. Nüwa has been evaluated on freshly built images and on real-world images from the IBM Research Compute Cloud (RC2), a compute cloud used by IBM researchers worldwide. When applying security patches to a fresh installation of Ubuntu-8.04, Nüwa successfully applies 402 of 406 patches. It speeds up the patching process by more than 4 times compared to the online approach and by another 2–10 times when integrated with Mirage. Nüwa also successfully applies the 10 latest security updates to all VM images in RC2.

1. INTRODUCTION

Patching is a basic and effective mechanism for computer systems to defend against most, although not all, security threats, such as viruses, rootkits, and worms [13, 19, 21]. Failing to promptly patch physical machines can subject the systems to huge risks, such as loss of confidential data, compromise of system integrity, and failure to provide regular system services. Unfortunately, applying security patches is a notoriously tedious task, due to the large number of patches and the high rate at which they are released — it is estimated that, in an average week, vendors and security organizations release about 150 vulnerabilities and associated patching information [15]. As a result, most software runs with outdated patches [11, 12].

The problem is exacerbated by the IT industry's recent shift to virtualization and cloud computing. Virtualization allows a complete system state to be conveniently encapsulated in a virtual machine (VM) image, which can run on any compatible hypervisor. Based on virtualization, cloud computing services (e.g., Amazon Elastic Compute Cloud (EC2) [2], NCSU Virtual Computing Lab (VCL) [20]) provide on-demand computing resources to customers' workloads, usually encapsulated in VM images. Because VM images are normal files, they can be easily copied to create new VM images. This has led to a new "VM image sprawl" problem, where a large number of VM images are created by the users and left unattended. A direct result of the VM image sprawl problem is the significantly increased management cost of maintaining these VM images, including regularly applying security patches to both active VMs and dormant VM images.

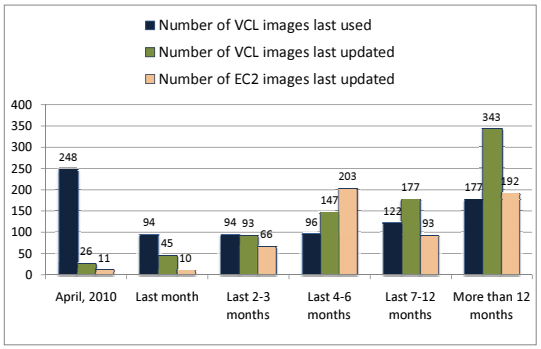


Figure 1: Counts of EC2 and VCL images, grouped by their time of last update or use (data collected on April 15, 2010)

A Glance at Two Compute Clouds: Figure 1 shows how recently VM images in two operational compute clouds, Amazon EC2 [2] and VCL [20], were updated; for VCL, the figure also shows how recently images were used. [1] There are a total of 831

[1]The VCL data was provided by the VCL management team, while the EC2 data was retrieved from the public Amazon Machine Images (AMIs) listed at Amazon's AMI page (`http://developer.amazonwebservices.com/connect/`

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

VM images in VCL and 575 public VM images posted at EC2's AMI page. However, more than 91% of the VCL images and more than 96% of the EC2 images have not been updated for at least 1.5 months. Moreover, more than 58% of the VCL images have not been used in the last 1.5 months. Note that it is not the case that these inactive images will not be needed in the future. Indeed, based on the VCL log, VCL purged 776 VM images marked by the users as "deleted" in the past; all of the remaining 831 images were explicitly marked as needed by their owners.

Our investigation of EC2 and VCL leads to two observations:

- Most VM images in compute clouds are not properly patched. The longer a VM image remains unpatched, particularly after a major vulnerability is discovered, the more likely it is to threaten other machines in the compute cloud or in the Internet. Also, unpatched images owned by organizations or companies may not be compliant with the organizations' security policies.
- A significant portion of the VM images are mostly offline and infrequently booted. Thus, any attempt to start these VMs and install patches will be an extra cost to the image owners. The cloud service providers may certainly offer patching as a free service; however, they will have to sacrifice CPU cycles that could potentially bring in revenues.

Inadequacy of Existing Patching Utilities: Traditional patching utilities, originally designed for running systems, require the VM images to be online before the patch can be applied. There are a few recent attempts to patch offline VM images using traditional patching utilities. For example, the Microsoft Offline Virtual Machine Servicing Tool [10] brings the VM image online, applies the patches, and captures the VM back to a dormant image. Realizing that not all VM images are needed immediately by their users, a lazy patching approach was developed in [27], which injects the patch installer and patch data into the VM image in such a way that the patch process is triggered at the next booting time. This optimization can yield significant savings in the total time spent in patching in the case where only a small percentage of dormant images will ever be used. However, the tradeoff is that users will now see delays in image startup, which can be significant for images that have accumulated a long list of yet-to-be-applied patches. Our own experiences show that update time can be fairly long (in the order of 10s of minutes) for stale systems (e.g., dormant for 1 month). In modern clouds where VM instances are dynamically provisioned to meet varying demands, this delay is unacceptable. Additionally, for enterprises systems, it is often required that all IT assets (physical or virtual, dormant or online) be up to date with regard to patches for security or compliance reasons. This will apply to cloud providers as enterprises embrace the cloud computing model. Finally, in a cloud environment where customers are charged for resources used during patching, this approach imposes costs that customers might not accept.

In general, patching approaches that require VMs to be online are a poor fit for VM images in compute clouds. Note that it takes on the order of minutes just to power up and shut down a VM image. With the large number of dormant VM images that are infrequently used, these approaches add significant extra costs either for customers or for cloud service providers. In addition to these costs, bringing a VM image online necessarily runs code that has nothing to do with patching, which makes patching less predictable.

`kbcategory.jspa?categoryID=171`). There are indeed more public AMIs in EC2 (more than 7,000 in US East, US West, and EU West EC2 sites in mid April 2010) than those in this list. Amazon does not publish usage data.

Our Solution–Nüwa Offline Patching Tool: We propose an approach that is fundamentally different from the traditional online model. We argue that the only way to make the patching process scalable in a cloud environment, where the number of images can potentially reach millions [2], is to do it offline. A closer look into the patching process reveals that it can be decomposed into a sequence of actions, not all of which require a running system. In fact, most of the patching actions only depend on and have an impact on file system objects, which are already encapsulated in the VM image itself. Among the actions that do depend on or have impacts on a running system, we find that many are unnecessary when patching offline, and some can be safely replaced by other actions that do not need the running system. Based on these findings, we design and implement Nüwa [3], a scalable offline patching tool for VM images. By patching offline, Nüwa avoids the expensive VM start and stop time, and, for the majority of cases, ensures that, when a VM image is ready to be started, it has the latest patches installed.

Because Nüwa is an offline patching tool, it can leverage novel VM image manipulation technologies to further improve scalability. In particular, Nüwa is integrated with the Mirage image library [24], which stores identical files once and treats images as logical views on this collection of files. By exploiting Mirage, Nüwa can patch all images that contain a file by patching that single file and updating each image's view, thus providing efficient and scalable offline patching in batch.

Our implementation of Nüwa supports the Debian package manager [5] and the RPM package manager [8]. We evaluated Nüwa with 406 patches to a freshly installed Ubuntu-8.04. Our evaluation shows that Nüwa applies 402 of the 406 patches offline and speeds up the patching process by more than 4 times compared to the online approach. This can be further improved by another 2–10 times when the tool is integrated with Mirage, making Nüwa an order of magnitude more efficient than the online approach. We also evaluated Nüwa on real-world images from the IBM Research Compute Cloud (RC2) [25], a compute cloud used by IBM researchers worldwide. Nüwa successfully applies the 10 latest security updates to all VM images in RC2.

This paper is organized as follows. Section 2 gives background information on patching and describes our design choices and technical challenges. Section 3 presents an overview of our approach. Section 4 describes the mechanisms we use to convert an online patch into one that can be safely applied offline. Section 5 describes how we leverage efficient image manipulation mechanisms to further improve scalability. Section 6 presents our experimental evaluation results. Section 7 discusses related work. Section 8 concludes this paper with an outlook to the future.

2. PROBLEM STATEMENT

2.1 Background

Software patches, or simply patches, are often distributed in the form of software update packages (e.g., *.deb* or *.rpm* files), which are installed using a package installer, such as `dpkg` and `rpm`. In this section, we give background information on the format of software packages and the package installation process. We use the Debian package management tool `dpkg` as an example. Most software package management tools follow the same general style with only slight differences.

[2]Amazon EC2 already contains over 7,000 public VM images as of April 2010, without including private images that users choose not to share with others [18].

[3]Named after the Chinese Goddess who patches the sky.

Packages are distribution units of specific software. A package usually includes files for different purposes and associated metadata, such as the name, version, dependences, description and concrete instructions on how to install and uninstall this specific software. Different platforms may use different package formats to distribute software to their users. But the contents are mostly the same. A Debian package, for example, is a standard Unix `ar` archive, composed of two compressed tar archives, one for the filesystem tree data and the other for associated metadata for controlling purposes. Inside the metadata, a Debian package includes a list of configuration files, md5 sums for each file in the first archive, name and version information, and shell scripts that the package installer runs at specific points in the package lifecycle.

The main action in patching is to replace the old buggy filesystem data with the updated counterparts. Moreover, the package installer also needs to perform additional operations to ensure the updated software will work well in the target environment. For example, dependences and conflicts must be resolved, a new user or group might have to be added, configuration modifications by the user should be kept, other software packages dependent on this one may need to be notified, and running instances of this software may need to be restarted. Most of these actions are specified in scripts provided by the package developers. Because these scripts are intended to be invoked at certain points during the patching process, they are called *hook scripts*. The hook scripts that are invoked before (or after) file replacement operations are called *pre-installation* (or *post-installation*) *scripts*. There are also scripts intended to be invoked when relevant packages (e.g., dependent software) are installed or removed.

More details about Debian package management tools can be found in the Debian Policy Manual [6].

2.2 Design Choices and Technical Challenges

Our goal is to build a patching tool that can take *existing* patches intended for online systems and apply them *offline* to a large collection of dormant VM images in a manner that is *safe* and *scalable*. By safety we mean that applying the patch offline achieves the same effect on the persistent file systems in the images as applying it online. By scalability we mean that the tool has to scale to thousands, if not millions of VM images. In this paper we only consider dormant VM images that are completely shutdown; VM images that contain suspended VMs are out of the scope of this paper.

We made a conscious design decision to be backward compatible with an existing patch format. It is tempting to go with a "clean slate" approach, where we define a new VM-friendly patch format and associated tools that do not make the assumption of a running system at the time of patch application. While this is indeed our long-term research goal, we think its adoption will likely take a long time, given the long history of the traditional online patching model and the fact that it is an entrenched part of today's IT practices, ranging from software development and distribution to system administration. Thus, we believe that an interim solution that is backward compatible with existing patch format, and yet works in an offline manner and provides much improved scalability, would be desirable.

Several technical challenges arise in developing such a scalable offline patching tool, as discussed below:

Identifying Runtime Dependences: The current software industry is centered around running systems and so are the available patching solutions. A running system provides a convenient environment to execute the installation scripts in the patch. The installation scripts query the configuration of the running system to customize the patch appropriately for the system. Some scripts also restart the patched software at the end of the patching process to ensure its effect takes place. Some patches require running daemons. For example, some software stores configuration data in a database. A patch that changes the configuration requires the database server to be running in order to perform schema updates.

The challenge is to separate runtime dependences that can be safely emulated (such as information discovery that only depends on the file system state) or removed (such as restarting the patched software) from the ones that cannot (such as starting a database server to do schema updates). We address this challenge by a combination of manual inspection of commands commonly used in scripts (performed only once before any offline patching) and static analysis of the scripts.

Removing Runtime Dependencies: Once we identify runtime dependences that can be safely emulated or removed, the next challenge is to safely remove these dependences so that the patch can be applied to a VM image offline and in a manner that does not break backward compatibility. Our solution uses a script rewriting approach that preserves the patch format and allows a patch intended for an online system to be applied safely offline in an emulated environment.

Patching at a Massive Scale: As the adoption of virtualization and cloud computing accelerates, it is a matter of time before a cloud administrator is confronted with a collection of thousands, if not millions of VM images. Just moving from online to offline patching is not sufficient to scale to image libraries of that magnitude. We address this challenge by leveraging Mirage's capabilities in efficient storage and manipulation of VM images [24].

3. APPROACH

It seems plausible that patching VM images offline would work, given the fact that the goal of patching is mainly to replace old software components, represented as files in the file system, with new ones. Indeed, to patch an offline VM image, we only care about the changes made to the file system in the VM image; many changes intended for a running system do not contribute to the VM image directly.

Simple Emulation-based Patching: One straightforward approach is to perform the file replacement actions from another host, referred to as the *patching host*. The patching host can mount and access an offline VM image as a part of its own file system. Using the `chroot` system call to change the root file system to the mount point, the patching host can emulate an environment required by the patching process on a running VM and perform the file system actions originally developed for patching a running VM. We call this approach *simple emulation-based patching* and the environment set up by the above procedure the *emulated environment*.

Failures and Observations: Unfortunately, our investigation shows that the installation scripts used by the patching process pose a great challenge to simple emulation-based patching. For example, Figure 2 shows two segments of code from `dbus.postinst`, the post-installation script in the `dbus` package. The first segment (lines 1 to 7) detects possibly running `dbus` processes and sends a reboot notification to the system if there exists one. The second segment (lines 9 to 16) restarts the patched `dbus` daemon so that the system begins to use the updated software. Both segments depend on a running VM to work correctly. The simple emulation-based patching will fail when coming across this script.

We looked into the internals of patching scripts. After analyzing patching scripts in more than one thousand patching instances, we made some important observations. First, most commands used in the patching scripts are *safe* to execute in the emulated environment, in the sense that *they do not generate undesirable side*

```
1 if [ "$1" = "configure" ]; then
2  if [ -e /var/run/dbus/pid ] &&
3     ps -p $(cat /var/run/dbus/pid); then
4     /usr/share/update-notifier/notify-reboot-required
5     ...
6  fi
7 fi
8 ...
9 if [ -x "/etc/init.d/dbus" ]; then
10  update-rc.d dbus start 12 2 3 4 5 . stop 88 1 .
11  if [ -x "`which invoke-rc.d`" ]; then
12     invoke-rc.d dbus start
13  else
14     /etc/init.d/dbus start
15  fi
16 fi
```

Figure 2: Excerpts of the `dbus.postinst` script

effects on the persistent file system that would make the patched VM image different from one patched online except for log files and timestamps. Examples of such commands include the test commands in lines 2, 9 and 11, `cat` in line 3, `/usr/share/update-notifier/notify-reboot-required` in line 4, `update-rc.d` in line 10, and `which` in line 11. Second, some command executions have no impact on the offline patching and thus can be skipped. For example, `invoke-rc.d` in line 12 of Figure 2 is supposed to start up a running daemon, and its execution has no impact on the persistent file system. Thus, we can just skip it. We call such code *unnecessary code*. Third, there are usually more than one way to achieve the same purpose. Thus, it is possible to replace an unsafe command with a safe one to achieve the same effect. For example, many scripts use `uname -m` to get the machine architecture; unfortunately, `uname -m` returns the architecture of the patching host, which is not necessarily the architecture for which the VM image is intended. We can achieve the same purpose by looking at the file system data, for example, the architecture information in the ELF header of a binary file.

Safety Analysis and Script Rewriting: Motivated by the above observations, in this paper, we propose a systematic approach that combines safety analysis and script rewriting techniques to address the challenge posed by scripts. The safety analysis examines whether it is safe to execute a script in the emulated environment, while the rewriting techniques modify unsafe scripts to either eliminate unsafe and unnecessary code, or replace unsafe code with safe one that achieves the same purpose. Our experience in this research indicates that the majority of unsafe scripts can be rewritten into safe ones, and thus enable patches to be applied to offline VM images in the emulated environment.

However, not all scripts can be handled successfully in this way. We find some patching instances, after safety analysis and rewriting, still unsafe in the emulation-based environment. Some patches have requirements that can only be handled in a running environment. For example, the post-installation script in a patch for MySQL may need to start a transaction to update the administrative tables of the patched server. As another example, `mono`, the open source implementation of C# and the Common Language Runtime, depends on a running environment to apply the update to itself.

The Nüwa Approach: To address this problem, we adopt a hybrid approach in the development of Nüwa. When presented with a patch, Nüwa first performs safety analysis on the patching scripts included in the original patch. If all scripts are safe, Nüwa uses simple emulation-based patching directly to perform offline patching. If some scripts are unsafe, Nüwa applies various rewriting techniques, which will be discussed in detail in Section 4, to these scripts, and performs safety analysis on the rewritten scripts. If these rewriting techniques can successfully convert the unsafe scripts to safe ones, Nüwa will use simple emulation-based patching with the rewritten patch to finish offline patching. However, in the worst case, Nüwa may fail to derive safe scripts through rewriting, and will resort to online patching. In reality, we have found such cases to be rare – our results show that less than 1% of the packages tested in our experiments fall into this category (Section 6.1).

In addition to patching individual VM images, Nüwa also leverages VM image manipulation technologies to further improve scalability. In particular, Nüwa uses features of the Mirage image library [24] to enable scalable patching of a large number of VM images in batch.

To distinguish between the two variations of Nüwa, we refer to the former as *standalone Nüwa*, and the latter, which leverages Mirage, as *Mirage-based Nüwa*. In the following, we describe the novel techniques developed for offline patching in the context of both standalone and Mirage-based Nüwa.

4. SCRIPT ANALYSIS AND REWRITING

This section explains how safe patch scripts are identified and, when possible, unsafe scripts are transformed into safe scripts. The analysis is based on three concepts — impact, dependence, and command classification, which are defined in Section 4.1. Section 4.2 presents rewriting techniques that, using information from safety analyses, convert many unsafe scripts into safe scripts.

In our implementation, safety analysis and script-rewriting run immediately before the package manager (i.e., `dpkg` and `rpm`) executes a patch script. As a result, analyses and transformations have access to the script's actual environment and arguments and to the image's filesystem state.

Patch scripts are in general shell scripts. For example, patch scripts in Debian are SUSv3 Shell Command Language scripts [17] with three additional features mandated by the Debian Policy Manual [6]. Patch scripts are executed by an interpreter that repeatedly reads a command line, expands it according to a number of expansion and quoting rules into a command and arguments, executes the command on the arguments, and collects the execution's output and exit status. The language is very dynamic (for example, command-lines are constructed and parsed dynamically), which forces our analyses and transformations to be conservative. Nonetheless, simple, syntax-directed analyses and rewritings suffice to convert unsafe scripts to safe versions for 99% of the packages we considered.

4.1 Impact, Dependence, and Command Classification

The goal of command classification is to divide a script's command lines into three categories: (1) safe to execute offline, (2) unsafe to execute offline, and (3) unnecessary to execute offline. To classify command lines, we divide a running system into a "memory" part and a "filesystem" part, and determine which parts may influence or be influenced by a given command line. The intuition is that the "filesystem" part is available offline but the "memory" part requires a running instance of the image that is being patched.

Table 1: Commands w/ FS-only impacts

Command Type	Example Commands
File attribute mod.	`chown, chmod, chgrp, touch`
Explicit file content mod.	`cp, mv, mknode, mktemp`
Implicit file content mod.	`adduser, addgrp, remove-shell`

We say that a command-line execution *depends on the filesystem* if it reads data from the filesystem or if any of its arguments or inputs flow from executions that depend on the filesystem. An

execution *impacts the filesystem* if it writes data to the filesystem or if its output or exit status flow to executions that impact the filesystem. Table 1 lists some commands whose executions impact the filesystem:

We say that a command-line execution *depends on memory* if it inspects any of a number of volatile components of the system's state (perhaps by listing running processes, opening a device, connecting to a daemon or network service, or reading a file under `/proc` that exposes kernel state) or any of its arguments or inputs flow from executions that depend on memory. An execution *impacts memory* if it makes a change to a volatile component of the system's state that outlives the execution itself, or if its output or exit status flow to executions that impact the memory.

Note that all executions have transient effects on volatile state: they allocate memory, create processes, cause the operating system to buffer filesystem data, and so forth. For the purposes of classification, we do not consider these effects to be impacts on memory; we assume that other command-line executions do not depend on these sorts of effects. Table 2 lists some commands that impact or depend on memory.

Table 2: Commands w/ memory impact/dependence

Command Type	Example Commands
Daemon start/stop	`invoke-rc.d, /etc/init.d/`
Process status	`ps, pidof, pgrep, lsof, kill`
System info. inquiry	`uname, lspci, laptop-detect`
Kernel module	`lsmod, modprobe`
Others	Database update, mono gac-install

The definitions for command-line executions are extended to definitions for static command lines. A command line depends on memory (or the filesystem) if any of its executions depend on memory (or the filesystem). A command line impacts memory (or the filesystem) if any of its executions impact memory (or the filesystem).

To seed impact and dependence analysis, we manually inspected all commands used in patch scripts to determine their intrinsic memory and filesystem impacts and dependences. This might seem to be an overwhelming task but, in practice, scripts use very few distinct commands; we found only about 200 distinct commands used by more than 1,000 packages. It may be possible to derive this information by instrumenting command executions. In practice, we expect that it would be provided by package maintainers.

Table 3: Command classification

Depend on FS	Depend on Memory	Impact on Memory	Impact on FS	Safety
Yes/No	No	No	Yes/No	Safe
Yes/No	No	Yes	Yes	Unsafe
Yes/No	Yes	No	Yes	Unsafe
Yes/No	Yes	Yes	Yes	Unsafe
Yes/No	No	Yes	No	Unnecessary
Yes/No	Yes	No	No	Unnecessary
Yes/No	Yes	Yes	No	Unnecessary

Our analysis concludes that a static command-line depends on memory if one of the following holds: (1) The command is unknown; (2) the command has an intrinsic memory dependence; (3) one or more of the arguments is a variable substitution; (4) the input is piped from a command that depends on memory; or (5) the input is redirected from a device, a file under `/proc`, or from a variable substitution.

The rules for filesystem dependences and for impacts are similar. Note that the analysis errs on the side of finding spurious dependences and impacts. That is, these analyses are simple "may-depend/may-impact" analyses, which are both flow and context insensitive.

Table 3 shows how each command line's classification as safe, unsafe, or unnecessary is determined from its filesystem and memory impacts and dependences. Safe command lines do not depend on or impact memory. These are the commands that can and should be executed offline. Script rewriting preserves these commands. Unnecessary command lines have no impact on the filesystem. There is no reason to execute them offline because they do not change the image. In fact, if they depend on or impact memory, then they must be removed because they might fail without a running instance. Script rewriting removes these commands. Unsafe command lines may execute incorrectly offline because they depend on or impact memory and also impact the filesystem. In some cases, script rewriting cannot remove these command lines because their filesystem impacts are required. If any unsafe command line cannot be removed, then the patch cannot be executed offline.

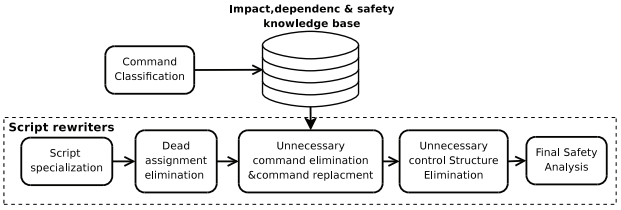


Figure 3: Flow of script analysis and rewriting

4.2 Rewriting Techniques

Figure 3 shows the rewriting techniques that Nüwa applies before executing each patch script. Rewriting a script can change the results of safety analysis, so Nüwa reruns safety analysis after applying these techniques. If safety analysis proves that all command lines in the script are safe, then the rewritten script is executed offline. Otherwise, Nüwa resorts to online patching.

Nüwa currently applies five rewriting techniques, which are described below. For clarity, the presentation does not follow the order in which the techniques are applied (that order is shown in Figure 3). The first two techniques consider command-lines, annotated by safety analysis, in isolation; the last three analyze larger scopes.

Unnecessary Command Elimination: This technique removes unnecessary commands, which, by definition, have neither direct nor indirect impact on the filesystem. Figure 4 shows an example.

```
/etc/init.d/acpid
/etc/init.d/cupsys
killall
```

Figure 4: Examples of command lines that are removed by unnecessary command elimination

Command Replacement: Some command lines that depend on memory can be replaced with command lines that depend only on the filesystem. This often happens with commands that need information about the system, in particular when the information is available both in the filesystem and, if there is a running instance, in memory.

```
uname -m
  -> dpkg --print-architecture
```

```
uname -s
  -> echo "Linux"
```

Figure 5: Memory-dependent command lines and their replacements

For example, the `uname` command prints system information; depending on its arguments, it will print the hostname, the machine hardware name, the operating system name, or other fields. `uname` gets its information from the kernel through the `uname` system call. Without a running instance, information from the kernel cannot be trusted. However, certain fields are statically known constants or available through commands that depend only on the filesystem; Figure 5 shows two examples.

Note that the command replacement technique not only removes memory-dependent commands but also ensures that the offline script uses values appropriate to the image instead of values from the host. Nüwa's implementation of command replacement consults a manually constructed table of command lines and their known replacements.

Before rewriting:

```
1  if [ -x "`which invoke-rc.d`" ]; then
2     invoke-rc.d dbus start
3  else
4     /etc/init.d/dbus start
5  fi
```

After rewriting:

```
All eliminated
```

Figure 6: Example of control structure analysis (from `dbus.postinst`)

Unnecessary Control-structure Elimination: This technique, a generalization of unnecessary command elimination, removes compound commands like `if` and `case` statements.

Figure 6 shows an example. Both the true branch and the false branch of the `if`-statement are unnecessary and would be eliminated by unnecessary command elimination. The conditional would not be eliminated because safety analysis conservatively assumes that all conditionals impact both memory and the filesystem through control-flow. By contrast, unnecessary control-structure elimination eliminates the entire `if`-statement because, after eliminating both branches of the `if`-statement, the conditional is unnecessary: It clearly has no filesystem impact through control-flow or any other means.

We perform unnecessary control-structure elimination in a bottom-up fashion (i.e., process inner control structures first). For each control structure being processed, we first try to eliminate all statements in each branch of the structure. If all statements in every branch can be eliminated, we consider the conditional itself: If it no longer impacts the filesystem, the entire control structure is removed.

Note that Nüwa applies unnecessary control-structure analysis to many kinds of compound commands and command lists, including the `case` construct and command lists built from the short-circuiting statements (`||` and `&&`).

Script Specialization: This technique removes command lines and control structures that cannot execute, given the script's actual environment and arguments and the VM image's filesystem state. Recall that this context is available because safety analysis and script-rewriting run immediately before `dpkg` executes a patch script.

Figure 7 shows an example, which was extracted from the post-installation script for the `acpid` package. Except during error recovery, `dpkg` calls post-installation scripts with `configure` as the first positional parameter (`$1`). Therefore, the `case` statement can be replaced with the first branch. Next, since the rest of the script changes neither `/var/run/hald/hald.pid` nor `/etc/init.d/hal`, the conditional can be evaluated at rewriting time; in this case, the conditional is false and the false branch is empty so the entire if statement is removed.

The current implementation of script specialization is a collection of ad hoc rewriting passes, which Nüwa applies before applying any other rewriting techniques. One pass replaces positional parameters with actual parameters. Another evaluates conditionals built from filesystem tests, when the tests depend only on the initial filesystem state. A third evaluates the command line `dpkg --compare-versions`, which is used frequently and whose result can be determined from the VM image's package database.

Before rewriting:

```
1  HAL_NEEDS_RESTARTING=no
2  case "$1" in
3    configure)
4      if [ -x /etc/init.d/hal ] &&
5        [ -f /var/run/hald/hald.pid ]; then
6        HAL_NEEDS_RESTARTING=yes
7        invoke-rc.d hal stop
8      fi
9      ;;
10   reconfigure)
11     ...
12 esac
```

After rewriting:

```
HAL_NEEDS_RESTARTING=no
```

Figure 7: Example of script specialization (from `acpid.postinst`)

All passes are conservative and err on the side of missing rewriting opportunities. For example, positional-parameter replacement leaves the script unchanged if the script uses the `shift` statement, which renames the positional parameters.

Dead-assignment Elimination: This technique removes assignments to unused variables. Some dead assignments come from the original scripts; others are created by script specialization, which can convert conditionally dead assignments to dead assignments.

Figure 8 shows an example of dead assignment, extracted from `xfonts-scalable.postinst`. In this script, the command `laptop-detect` is intrinsically memory-dependent. If its result flows to a command line that impacts the filesystem, the script would be unsafe. Fortunately, the `LAPTOP` variable is unused in the rest of the script. Removing its assignment leaves the body of the inner `if` statement empty, which makes the conditional unnecessary, which in turn allows the entire inner `if` statement to be removed. The outer `if` statement is then removed in a similar fashion.

Before rewriting:

```
LAPTOP=""
if [ -n "$(which laptop-detect)" ]; then
    if laptop-detect >/dev/null; then
            LAPTOP=true
    fi
fi
```

After rewriting:

```
All eliminated
```

Figure 8: Example of dead-assignment elimination

The first assignment in Figure 7, which is conditionally dead in the original script, could be transformed into a dead assignment by script specialization.

Dead-assignment elimination depends on a syntax-directed data-flow analysis of the main body of the script. An assignment is *dead* if the assigned value cannot reach a *use* before reaching the end of the script or another assignment; the analysis conservatively judges an assignment to be dead if it it does not occur in a loop and is followed by another assignment in the same syntactic scope, with no intervening uses in any syntactic scope, or if no uses follow at all. Function bodies are not considered, except that any use of a variable within a function body is considered reachable from any assignment to that variable in the entire program.

5. SCALABLE BATCH PATCHING

A motivating assumption of this work is that, as cloud computing becomes more widely adopted, image libraries will grow to contain thousands or perhaps even millions of images, many of which must be patched as new vulnerabilities are discovered. Even with the offline patching techniques presented in Section 4, patching so many images individually would take a significant amount of time.

This section explains an approach to batch patching a large number of images offline that exploits an observation and a conjecture about patching images. The observation is that, if the same patch is applied to two similar images, then any given patch-application step is likely to have the same effect on both images. For example, the same files will be extracted from the patch both times. The conjecture is that the images that must be patched are likely to be similar to one another; this conjecture seems particularly reasonable for clouds (such as Amazon's EC2 [2]) that encourage users to derive new images from a small set of base images.

Nüwa's batch patching harnesses Mirage, a scalable VM image storage solution that exploits the similarity between images [24]. We first give a brief overview of Mirage before describing the batch patching solution.

5.1 Overview of Mirage

The Mirage image library maintains a collection of VM images and provides an image-management interface to users: users can import images into the library, list existing images in the library, check out a particular image, and check in updates of the image to create either a new image or a new version of the original image. A separate interface allows system administrators to perform system-wide operations, such as backup, virus scan, and integrity verification of all image content.

A design goal of Mirage is to support operations on images as structured data. To this end, Mirage does not store images as simple disk images. Instead, when an image is imported into the library, Mirage iterates over the image's files, storing each file's contents as a separate item in a content-addressable store (CAS); the image as a whole is represented by a manifest that refers to file-content items and serves as a recipe for rebuilding the image when it is checked out. An earlier paper [24] described this format and explained how it allows certain operations on images to be expressed as fast operations on the image's manifest. For example, creating a file, assuming that the desired contents are already in the CAS, reduces to adding a few hundred bytes to the manifest.

Mirage's new *vmount* feature, which was not described in the earlier paper, allows users to mount library images without rebuilding them. Vmount is implemented as a FUSE [26] daemon and fetches data from the CAS as it is demanded; by contrast, checking out an image requires fetching every file's contents from the CAS. Vmount also implements a new extended filesystem attribute that allows direct manipulation of the manifest. For each regular file, the value of this attribute is a short, unique identifier of the file's contents. Setting the attribute atomically replaces the file's contents with new contents.

After modifying an image through Vmount, the user can check in the changes as a new image or as a new version of the original image. The original image is not disturbed, and the time to check in is proportional to the amount of new data instead of to the size of the image.

Vmount has three benefits for batch patching. First, there is no need to rebuild each image. Arguably, this is merely a workaround for a problem created by the decision to store images as manifests. Second, if two images share data in the CAS and are patched sequentially through Vmount, then reading the shared data the second time is likely to be fast, because the data will be in the host's buffer cache. By contrast, if two disk images are patched sequentially, then the fact that they share data is effectively hidden from the host's operating system. The largest benefit is that Vmount allows batch patching to operate on manifests without major modifications of system tools like `dpkg`. Time-critical patching steps can be changed to use the new filesystem attribute, without creating a dependence on the manifest format, while less profitable steps continue to use the normal filesystem interface.

5.2 Batch Patching via Mirage

A straightforward way to patch a batch of images is to iterate the patching process for individual images. For images in Mirage, each iteration mounts an image with Vmount, applies the patch [4], and checks in the modified image.

Our approach optimizes this straightforward approach by moving invariant operations out of the loop that visits each image. Currently, Nüwa optimizes one source of invariant operations: unpacking the patch, which copies the patch's files to the image and, ultimately, adds their contents to the Mirage CAS. These copies and CAS additions are good operations to move out of the loop because they consume most of the time of applying most patches; in future work, we plan to hoist more invariants out of the loop.

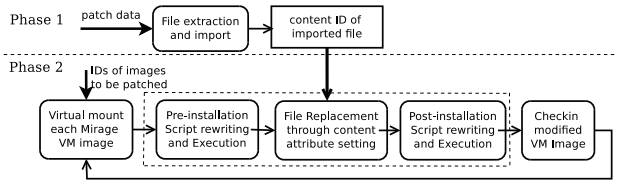


Figure 9: Batch patching VM images via Mirage

Figure 9 shows the two phases of batch patching via Mirage. Phase 1 performs the loop-invariant operation: Nüwa extracts the patch's files and imports them into Mirage. The result is a list of content identifiers, one for each file. In phase 2, Nüwa iterates over the images. For each image, Nüwa mounts the image with Vmount, rewrites and executes the pre-installation scripts, emulates the "unpack" step of the package manager (e.g., `dpkg`), using the Mirage filesystem attribute to set the contents of the patch's files, rewrites and executes the post-installation scripts, and checks in the modified VM image. If script rewriting ever fails to produce a safe script, then Nüwa resorts to online patching.

6. EXPERIMENTAL EVALUATION

We have implemented both standalone Nüwa and Mirage-based Nüwa by extending the Debian Almquist Shell (i.e., `dash`) [4]. (Our script rewriting was performed based on the syntax tree generated by `dash`.) Our implementations assume a Linux host sys-

[4]If the patch must be applied online, then the image must be rebuilt.

tem. We have tested the standalone Nüwa on patching hosts running CentOS 5.2, Ubuntu 9.0.4 and OpenSuSE 11.1. Our implementations currently support VM images of any Linux distributions based on Debian package management tools (e.g., Debian, Ubuntu, Knoppix) or RPM package manager (e.g., RHEL, CentOS). However, Mirage-based Nüwa currently only works on the Debian package manager, as the optimizations have not been completely ported to RPM yet.

We performed three sets of experiments to evaluate Nüwa, including (1) patching individual VM images offline, (2) Mirage-based offline patching in batch, and (3) patching VM images in a real-world compute cloud RC2. The first two sets of experiments were performed on a DELL OptiPlex 960 PC, with a 3GHz Intel Core 2 Duo CPU and 4GB DDR2 memory. The third set of experiments were performed in RC2. Unless otherwise noted, we used the x86-64 version of OpenSuSE 11.1 version as the patching host OS in all experiments. For compatibility reasons, we updated its kernel to version `2.6.31.11-0.0.0.2.9c60380-default`.

6.1 Patching Individual VM Images

The objective of this set of experiments is two-fold: First, we would like to evaluate the correctness of the offline patching approach used in Nüwa (i.e., whether the offline patching approach has the same effect on the VM images as online patching). Second, we would like to see the efficiency of our offline patching approach in Nüwa compared with the online patching approach.

In this set of experiments, we used the Linux Kernel-based Virtual Machine (KVM) [7] to start instances of VM images for online patching. For offline patching, we used the VMware disk library to mount the VM images in the host environment. Our tool can be logically decomposed into two parts: the script rewriter and the patch applier. We copied both components into the mounted VM image, with the patch applier replacing the original package installer inside the target VM image.

To perform the evaluation, we first created an empty disk image in the flat VMDK disk format with the `kvm-img` image creation tool, then brought this disk image online through KVM, and installed a default configured 64-bit Ubuntu-8.04 inside. This was used as the base VM image for both offline and online patching.

We gathered all 406 patches available for the base VM image (64-bit Ubuntu-8.04) on October 26, 2009. The correctness of offline patching is verified by a file-by-file comparison of the results of online and offline: If two VM images, which are obtained through patching the base VM image online and offline, respectively, differ only in log files and timestamps, we consider the offline patching to be correct. To further evaluate the effectiveness of the rewriting techniques, we used the simple emulation-based patching mentioned in Section 3 as a reference.

Table 4: Comparison of offline patching methods

	# successes	# failures	success ratio
Simple emulation	369	37	90.9%
Nüwa	402	4	99.0%

Table 4 shows the experimental results for evaluating the correctness of our techniques. Nüwa can successfully apply 402 out of the 406 patches offline, achieving a 99.0% success ratio. The results also show that the rewriting techniques contributed significantly to the success; they helped improve the success ratio by about 10%. Note that the failure cases are failures of offline patching, not of Nüwa; Nüwa automatically detects all of these failures and can cope with them through automatic online patching, as discussed in Section 3.

The four failure cases are the `mono-gac` package[5] and three other packages that depend on `mono-gac`. Through further analysis, we found that `mono-gac` failed because the installer needed to access some kernel information (e.g., `/proc/self/map` and `/proc/cpuinfo`) in order to work correctly. This information cannot be retrieved in the emulated environment.

To compare the efficiency of Nüwa's offline patching techniques with that of online patching, we performed another set of experiments. We assumed the most efficient form of online patch, automated online patching. Specifically, we insert the patch data into the VM image through the emulated environment and then schedule a patching process at boot time by modifying the booting script in the VM image. We then boot the VM, perform online patching, and shut down the VM automatically once the patching is complete.

We collected two sets of data from these experiments. The first is the time (in seconds) required to apply each applicable patch to the base VM image through the offline patching approach in Nüwa, and the second is the time needed to apply the same set of patches through automated online patching.

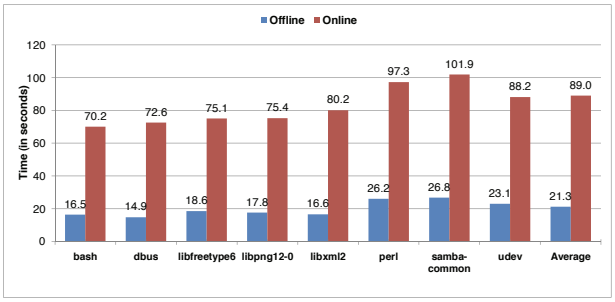


Figure 10: Time used by offline and online patching ("Average" is computed over 402 applicable packages)

Figure 10 shows the time (in seconds) required to apply some applicable patches to the base VM image through the Nüwa offline patching and the automated online patching, respectively. Due to the limited space, we only show the timing results for eight selected patches and the average for all 402 applicable patches. On average, the Nüwa offline approach takes only 23.9% of the time required by automated online patching (a factor of 4 speedup). This improvement, combined with the fact that Nüwa needs much less human intervention and physical resources, shows that it brings significant benefits to patching VM images.

This set of experiments demonstrates that Nüwa's offline patching techniques, particularly the rewriting techniques, are effective and that offline patching using Nüwa can significantly reduce the overhead involved in patching.

6.2 Batch Patching via Mirage

The primary objective of this set of experiments is to measure the scalability offered by Mirage-based Nüwa by comparing the performance of Mirage-based batch patching with that of one-by-one patching.

We generated 100 VM images using 32-bit Ubuntu 8.04 as the base operating system for this set of experiments. The Ubuntu installer can install a support for a number of basic, predefined tasks; some of these tasks are for running specific servers, while others are for desktop use. We generated test VM images from 100 randomly selected subsets of 12 of these tasks (listed in Table 5).

[5] `mono-gac` is a utility to maintain the global assembly cache of mono, an open source implementation of C# and the CLR.

Table 5: Basic Ubuntu tasks

#	Task Name	#	Task Name
1	lamp-server	2	mail-server
3	dns-server	4	openssh-server
5	print-server	6	samba-server
7	postgresql-server	8	ubuntustudio-audio
9	ubuntustudio-audio-plugins	10	ubuntustudio-graphics
11	ubuntustudio-video	12	ubuntu-desktop

We retrieved 154 security updates (i.e., security patches) for 32-bit Ubuntu 8.04 from Ubuntu Security Notices [28]. We also retrieved the ranking of Ubuntu packages given by Ubuntu's popularity contest [9], and sorted the 154 security patches accordingly. For our performance evalution, we selected the security updates corresponding to the eight most popular packages (as of January 18, 2010), including `dash, libdbus, libglib-2.0, libfreetype, udev, libpng, libxml2`, and `dbus`.

For each of the eight patches, we measured the time to apply the patch to the test VM images one-by-one and the time to apply the patch to the test VM images as batches of increasing sizes. Figure 11 shows that for all eight security patches, Mirage-Nüwa achieves considerable speedup over individual patching. Moreover, the speedup also increases as the number of images patched in a batch increases, and plateaus between 80 and 100 images.

For seven of the eight security patches (`udev` is the exception), the average speedup over one-by-one patching increases from 5.1 times to 8.5 times as the number of images in a batch increases from 10 to 100. Note that this speedup is on top of the factor of 4 speedup achieved over traditional online patching, thus bringing the total speedup over traditional online patching to about 30 when patching 100 images in a batch.

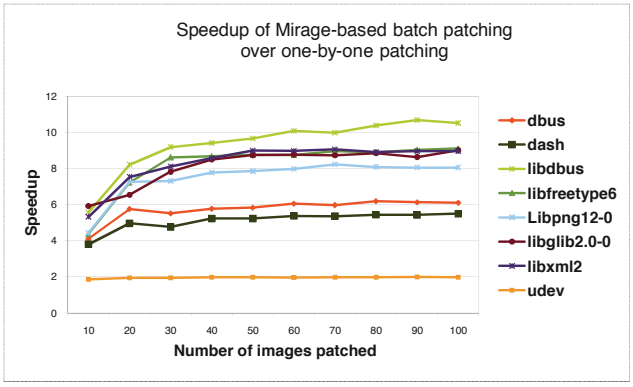


Figure 11: Scalability of Mirage-based Nüwa

However, the speedup for `udev` is much smaller, compared with the other seven patches. In fact, the speedup for `udev` is only around 2. Further investigation showed that the `udev` patch spends more time in pre-installation and post-installation scripts than the others; thus, the file replacement operations constitute a smaller portion of the entire patching process.

This set of experiments demonstrates that Mirage-based Nüwa is scalable and can improve the performance of offline patching significantly. Overall, Nüwa offline patching is an order of magnitude more efficient than online patching.

6.3 Patching a Real Cloud

In this experiment we assess the performance of Nüwa in a real production cloud. We patch the entire repository with the latest security updates published in the OS distributor's website. We set out to answer two questions: 1) how many of the images can be successfully patched offline using Nüwa, and 2) whether it is feasible to patch the *entire* image repository on a *daily* basis.

Our experiments are based on RC2 [25], a compute cloud very similar to Amazon's EC2, that is used by IBM researchers worldwide. Although small compared to EC2, RC2 is a production cloud that is used daily by IBM researchers. We created a replica of the RC2 image repository in our own testbed, so as to have a controlled experimental environment. The replica contains a total of 278 images, to which we apply the security patches from Red Hat's security advisories website [23]. All 278 images are running Red Hat 5.3 with the exception of one that is running CentOS 5.3. We used the RPM-based implementation of Nüwa since all Red Hat distributions use RPM for package installation. For this experiment we did not use the Mirage batch optimization because this feature has not yet been implemented in the RPM-based Nüwa.

We set up a dedicated host to run the patch process. The host is a blade with 4 Xeon 3.16GHz processors and 8GB RAM, running OpenSuse 11.1. The image repository is on a different, similar blade and the host accesses the repository via an NFS mount through a SAN network.

Table 6: Latest applicable security updates from RedHat rated "important" and higher

#	Update	Severity	Advisory
1	krb5	critical	RHSA-2010:0029
2	nspr/nss	critical	RHSA-2009:1186
3	openssl	important	RHSA-2010:0162
4	sudo	important	RHSA-2010:0122
5	acpid	important	RHSA-2009:1642
6	elinks	important	RHSA-2009:1471
7	dnsmasq	important	RHSA-2009:1238
8	bind	important	RHSA-2009:1179
9	cups	important	RHSA-2009:1082
10	freetype	important	RHSA-2009:1061

Patching the entire repository of 278 available images with the latest critical security update (krb5 [22]) takes about 45 seconds per image, totaling about 3.5 hours. All images were patched successfully, completely offline. Note that the patching time includes all time to set up the image for patching, download the update from a remote Red Hat Network server, and install the downloaded packages. We believe this number can be reduced 10-fold with an optimized storage configuration (e.g., a repository on a local disk or on direct-attached SAN storage), a local package server, and the Mirage batch-patching optimization, thus potentially allowing an average compute node (which can itself be a VM in the compute cloud) to apply a single security patch to about 19,200 (24*3600S/(45S/10)) images on a daily basis.

To test the robustness of Nüwa, we took the latest applicable security updates (shown in Table 6) from Red Hat's security advisories [23] that are rated "important" or "critical" and applied them across the entire repository. There are 10 updates which consist of 24 individual packages. All updates were successful on all 278 images, suggesting that Nüwa is robust enough to be offered as a real service in a production cloud.

7. RELATED WORK

Several available commercial tools [10,27,29] can apply patches to dormant VM images. But that does not mean the patches are applied in an *offline* manner. As a matter of fact, all of them require the image to be running when the patches are actually installed. Microsoft's Offline VM Servicing Tool [10] first "wakes" up the virtual machine (deploys it to a host and starts it), then triggers the appropriate software update cycle to apply the patches, and finally shuts down the updated virtual machine and returns it to the image

library. In the cases of VMware Update Manager [29] and Shavlik NetChk Protect [27], patches are first inserted into image at some specified locations, then applied when the image is powered up. We resort to this approach when Nüwa identifies patches that contain unsafe commands.

In some cases, it is preferable to apply patches online. In general, systems that tend to stay online for a long period of time, such as highly available servers, fall into this category. In those cases, "dynamic update" techniques [1,3,14,16] are used to apply patches to the target software without shutting them down. In contrast, Nüwa targets VM images that have already been shut down and may stay in dormant state for an extended period of time. Thus, these approaches are complimentary to Nüwa.

8. CONCLUSION

In this paper, we developed a novel tool named Nüwa to enable efficient patching of offline VM images. Nüwa uses safety analysis and script rewriting techniques to convert patches, or more specifically the installation scripts contained in patches, which were originally developed for online updating, into a form that can be applied to VM images offline. Nüwa also leverages the VM image manipulation technologies offered by the Mirage image library [24] to provide an efficient and scalable way to patch VM images in batch. We implemented a standalone Nüwa and a Mirage-based Nüwa; standalone Nüwa supports two popular package managers, the Debian package manager [5] and the RPM package manager [8], while Mirage-based Nüwa supports only the former. In addition to evaluating Nüwa with security patches and VM images configured with popular packages according to Ubuntu popularity contest, we also applied Nüwa to a real cloud RC2. Our experimental results demonstrate that 1) Nüwa's safety analysis and script rewriting techniques are effective – Nüwa is able to convert more than 99% of the patches to safe versions that can then be applied offline to VM images; 2) the combination of offline patching with additional optimization made possible through Mirage allows Nüwa to be an order of magnitude more efficient than online patching; and 3) Nüwa successfully patched 278 images in a real compute cloud.

A limitation of Nüwa is that it currently does not support offline patching of a suspended VM image, which includes a snapshot of the system memory state in addition to the file system. In our future research, we will investigate techniques to patch suspended VM images and how to perform offline patching on Windows platforms.

Acknowledgement

This work is supported by the U.S. National Science Foundation (NSF) under grant 0910767, and by an IBM Open Collaboration Faculty Award. The contents of this paper do not necessarily reflect the position or the policies of the U.S. Government or IBM.

9. REFERENCES

[1] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. Opus: online patches and updates for security. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 19–19, Berkeley, CA, USA, 2005. USENIX Association.

[2] Amazon. Amazon elastic compute cloud (EC2). `http://aws.amazon.com/ec2/`.

[3] Jeff Arnold and M. Frans Kaashoek. Ksplice: automatic rebootless kernel updates. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 187–198, New York, NY, USA, 2009. ACM.

[4] Debian community. Debian Almquist shell. `http://en.wikipedia.org/wiki/Debian_Almquist_shell`.

[5] Debian community. Debian package manager. `http://www.debian.org/dpkg`.

[6] Debian Community. Debian policy manual. `http://www.debian.org/doc/debian-policy/`, 2009.

[7] KVM community. Linux kernel-based virtual machine. `http://www.linux-kvm.org/`.

[8] RPM community. RPM package manager. `http://www.rpm.org/`.

[9] Ubuntu Community. Ubuntu popularity contest. `http://popcon.ubuntu.com/`.

[10] Microsoft Corporation. Offline virtual machine servicing tool 2.1. `http://technet.microsoft.com/en-us/library/cc501231.aspx`.

[11] Forbes. Cybersecurity's patching problem. `http://www.forbes.com/2009/09/14/sans-institute-software-technology-security-cybersecurity.html`. Visited on 2009-11-06.

[12] Stefan Frei, Thomas Duebendorfer, Gunter Ollmann, and Martin May. Understanding the Web browser threat. Technical Report 288, TIK, ETH Zurich, June 2008. Presented at DefCon 16, Aug 2008, Las Vegas, USA. `http://www.techzoom.net/insecurity-iceberg`.

[13] Thomas Gerace and Huseyin Cavusoglu. The critical elements of the patch management process. *Commun. ACM*, 52(8):117–121, 2009.

[14] Deepak Gupta and Pankaj Jalote. On line software version change using state transfer between processes. *Softw. Pract. Exper.*, 23(9):949–964, 1993.

[15] Huseyin Cavusoglu Hasan, Hasan Cavusoglu, and Jun Zhang. Economics of security patch management. In *The Fifth Workshop on the Economics of Information Security (WEIS 2006)*, June 2006.

[16] Michael Hicks and Scott M. Nettles. Dynamic software updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1049–1096, November 2005.

[17] The IEEE and The Open Group. The single UNIX specification, version 3. `http://www.unix.org/version3/online.html`, 2004.

[18] Cloud Market. The cloud market: EC2 statistics. `http://thecloudmarket.com/stats`.

[19] Microsoft. The microsoft security update release cycle. `http://www.microsoft.com/security/msrc/whatwedo/updatecycle.aspx`.

[20] NC State University. NC State University virtual computing lab (VCL). `http://vcl.ncsu.edu/`.

[21] United States General Accounting Office. Effective patch management is critical to mitigating software vulnerabilities. gao-03-1138t, September 2003.

[22] RedHat. Critical: krb5 security update.

[23] RedHat. RedHat Security Advisories. `http://rhn.redhat.com/errata/rhel-server-errata-security.html`.

[24] D. Reimer, A. Thomas, G. Ammons, T. Mummert, B. Alpern, and V. Bala. Opening black boxes: using semantic information to combat virtual machine image sprawl. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 111–120, 2008.

[25] Kyung Dong Ryu, Xiaolan Zhang, Glenn Ammons, Vasanth Bala, Stefan Berger, Dilma M. Da Silva, Jim Doran, Frank Franco, Alexei Karve, Herb Lee, James A. Lindeman, Ajay Mohindra, Bob Oesterlin, Giovanni Pacifici, Dimitrios Pendarakis, Darrell Reimer, and Mariusz Sabath. RC2 – A Living Lab for Cloud Computing. In *Lisa '10: Proceedings of the 24th Large Installation System Administration*, 2010. Earlier version available as an IBM technical report at `http://domino.watson.ibm.com/library/CyberDig.nsf/Home`.

[26] Miklos Szeredi. Fuse: Filesystem in userspace. `http://fuse.sourceforge.net/`, 2010.

[27] Shavlik Technologies. Offline virtual machine image quick start guide. `http://www.shavlik.com/documents/qsg-prt-6-1-offline_vm.pdf`.

[28] Ubuntu. Ubuntu security notices. `http://www.ubuntu.com/usn/`.

[29] VMware. VMware vcenter update manager. `http://www.vmware.com/products/update-manager/`.

A Framework for Testing Hardware-Software Security Architectures*

Jeffrey S. Dwoskin
Princeton University
Princeton, NJ, 08540 USA
jdwoskin@princeton.edu

Mahadevan Gomathisankaran†
University of North Texas
Denton, TX, 76203 USA
mgomathi@unt.edu

Yu-Yuan Chen
Princeton University
Princeton, NJ, 08540 USA
yctwo@princeton.edu

Ruby B. Lee
Princeton University
Princeton, NJ, 08540 USA
rblee@princeton.edu

ABSTRACT

New security architectures are difficult to prototype and test at the design stage. Fine-grained monitoring of the interactions between hardware, the operating system and applications is required. We have designed and prototyped a testing framework, using virtualization, that can emulate the behavior of new hardware mechanisms in the virtual CPU and can perform a wide range of hardware and software attacks on the system under test.

Our testing framework provides APIs for monitoring hardware and software events in the system under test, launching attacks, and observing their effects. We demonstrate its use by testing the security properties of the Secret Protection (SP) architecture using a suite of attacks. We show two important lessons learned from the testing of the SP architecture that affect the design and implementation of the architecture. Our framework enables extensive testing of hardware-software security architectures, in a realistic and flexible environment, with good performance provided by virtualization.

1. INTRODUCTION

Designers of security architectures face the challenge of testing new designs to validate the required security properties. To provide strong guarantees of protection, it is often necessary and desirable to place low-level security mechanisms in the hardware or the operating system kernel, which the higher-level software layers can rely upon for a wide-range of applications. The resulting architecture is a combination of hardware, kernel, and application software components which are difficult to test together. The security of the system as a whole relies on the combination of the correct design and implementation of the low-level security features, the correct and secure use of those features by the software layers, and the security of the software components themselves. Therefore, we need a framework that can comprehensively model the architecture and study the interactions between hardware and software components, running a realistic software stack with a full OS and applications, during normal operation and under attack.

We propose a testing framework that can emulate the hardware components of a security architecture and can provide a controlled environment with a full software stack, with which coordinated security attacks can be performed and observed. We have designed our testing framework with the initial goal of verifying the SP (Secret Protection) architecture [13, 24], while being generalizable to other security architectures. SP places roots of trust in the hardware which are used to protect security-critical software at the application layer, skipping over the operating system layer in the trust chain. The threat model includes attacks on software components as well as physical attacks, with only the processor chip itself trusted. The operating system remains untrusted and essentially unmodified. The "layer-skipping" feature of SP's minimalist trust chain is in contrast to traditional hierarchical trust chains, and testing with a commodity OS is necessary to verify that security-critical applications can be built on this type of architecture.

The testing environment — including the hardware implementation, software stack, threat models, and attack mechanisms — must be as realistic as possible. As far as we know, no existing testing methods provide a fast and convenient way to test both hardware and software security mechanisms simultaneously, running an unmodified commodity OS with a full microprocessor and hardware system, with full observability and controllability of coordinated hardware, software and network attacks.

Furthermore, our framework allows testing to be done during the design time; this gives confidence in the architecture before the complete system is built, at which point it is costly to make fundamental changes in response to security flaws.

*This work was supported in part by NSF CCF-0917134 and NSF CNS-0430487 (co-sponsored by DARPA). Access to VMware was provided through the VMAP program.

†M. Gomathisankaran was a postdoc at Princeton for this work.

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

For example, we show two important lessons learned while testing the implementation of the SP architecture. Although this paper focuses on testing integrated hardware-software security architectures like SP, it is also useful for debugging and testing software-only architectures.

The primary contributions of this work are:

- a new flexible framework for design-time testing of the security properties of hardware-software architectures;
- enabling testing with a realistic software stack, using commodity operating systems, and different applications using the new security mechanisms;
- a flexible, fast, and low-cost method for emulating hardware security features, using virtualization, for the purpose of design validation — without costly and time-consuming fabrication of hardware prototypes;
- an improved architecture for SP's secure memory mechanism and its implementation; and
- the application of our framework toward the validation of the security properties of the SP architecture, by providing a suite of attacks on SP's security mechanisms, as well as general attacks on the system.

2. THREAT MODEL AND ASSUMPTIONS

We focus on hardware-software architectures where new hardware security mechanisms are added to a general-purpose computing platform to protect security-critical software and its critical data. The hardware in the architecture provides strong non-circumventable security protection, and the software provides flexibility to implement different security policies for specific applications and usage scenarios.

We assume a system with security-critical software applications running on a platform with new hardware security mechanisms added to the CPU (e.g., new instructions, registers, exceptions, and crypto engines). Sometimes the OS cannot be trusted, especially if it is a large monolithic OS like Windows or Linux. Other times, an architecture might trust parts of the operating system kernel (e.g., a microkernel [1]), but not the entire OS.

We consider three classes of attacks in our testing framework. First, malware or exploitable software vulnerabilities that can allow adversaries to take full control of the operating system to perform software attacks. They can access and modify all OS-level abstractions such as processes, virtual memory translations, file systems, system calls, kernel data structures, interrupt behavior and I/O.

Second, hardware attacks, which can be performed by adversaries with physical possession of a device, such as directly accessing data on the hard disk, probing physical memory, and intercepting data on the display and I/O buses. We can also model some software attacks as having the same impact as these physical attacks.

Third, network attacks that can be performed with either software or hardware access to the device, or with access to the network itself. Some network attack mechanisms act like software attacks (e.g., remote exploits on software), while others attack the network itself (e.g., eavesdropping attacks) or application-specific network protocols (e.g., modification attacks and man-in-the-middle attacks).

In order to adequately test a new security architecture, all of these attack mechanisms must be considered and tested, according to the threat model of the particular system. Our testing framework provides hooks into each relevant system component, and allows information and events at each level to be correlated to emulate the most knowledgeable attacker.

Overall, we consider the functional correctness of the new hardware security mechanisms and the security-critical software components, as well as the interaction between these hardware and software components. We do not consider timing or other side-channel attacks.

Buggy or malicious hardware is considered an orthogonal problem within the manufacturing process – and not part of our threat model. However, to the extent that the emulated system corresponds functionally to the real microarchitecture, our framework can be used to generate data for test cases to run against manufactured devices, or to provide inputs to other verification schemes.

3. TESTING FRAMEWORK

We first describe the overall architecture of our testing framework, followed by the technical details of the framework components. We then show the range of attacks and events we can model, and finally present our prototype implementation.

3.1 Architecture

We build our testing framework on top of existing virtualization technology, which allows us to run a full set of commodity software efficiently. A virtual machine monitor (VMM) is the software that creates and isolates Virtual Machines (VMs), efficiently providing an execution environment in each VM which is almost identical to the original machine [33, 35]. By modifying an existing VMM's hardware virtualization, we can augment the virtual machine to have the additional hardware features of a new security architecture. Using virtualization allows the unmodified hardware and software components to run at near-native speed, while permitting our framework to intercept events and system state as needed.

Our Testing Framework is divided into two systems, as shown in Figure 1: the System Under Test (SUT) and the Testing System (TS), each running as a virtual machine on our modified VMM. The SUT is meant to behave as closely as possible to a real system which has the new security architecture. It can invoke the new hardware security primitives, along with the associated protected software for that architecture. In our current system, the SUT runs a full commodity operating system (Linux) as its guest OS, which is vulnerable to attack and is untrusted.

The TS machine simulates the attacker, who is trying to violate the security properties of the SUT. It is kept as a separate virtual machine so that the TS Controller can be outside of the SUT to launch hardware attacks. The virtualization isolates all testing activity and networking from the host machine.

The testing framework itself is independent of the threat model of the system being tested, and hence enables full controllability and observability of the SUT in both hardware and software. This makes it suitable for many purposes during the design phase of a new architecture. During the initial design and implementation of the system, the TS can act as a debugger, able to see the low-level behavior in hardware, all code behavior, and data in the software stack. When testing the supposedly correct system, the TS is the attacker, constrained by a threat model to certain attack vectors.

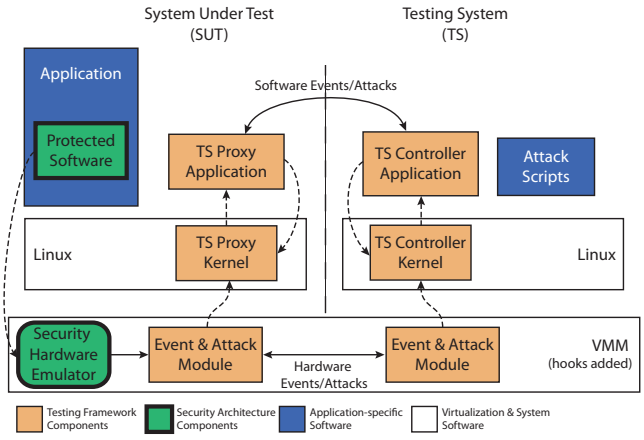


Figure 1: Testing Framework Design

A particular point of elegance of our framework is that the threat model can be easily changed, and the set of attack tools given to the attacker adjusted for each test. The framework can be used for arbitrary combinations of mechanisms: access to internal CPU state of the virtual processor, physical attacks on the virtual machine hardware (e.g. hardware probes on the buses, memory, or disk), software attacks on the operating system (e.g. a rootkit installed in the OS kernel), and network attacks (e.g. interception and modification of network packets and abuse of network protocols and application data). For example, in some cases, it might be desirable to perform black-box testing of a new design using only the network to gain access to the SUT, while in other cases, white-box testing will allow the attacker knowledge about the system's activities, such as precise timing of attacks with hardware interrupts or breakpoints into the application code, or observation of data structures in memory.

3.2 Testing Framework Components

The main components of our Testing Framework are shown in Figure 1. The framework detects events in the SUT and provides the TS with access to the full system state using both hardware and software channels. The TS Controller, running in the TS, is the aggregation point that receives events from both hardware and software. It receives OS and Application level (software) events from the SUT via a network channel and receives hardware events from the VMM. It provides APIs to the Attack Scripts which can monitor or wait for specific events and adaptively mount a coordinated attack on the SUT.

The TS Proxy is added to the SUT to communicate with the TS Controller to receive commands and send events back. It simulates the effect of a compromised operating system for launching software attacks, allowing the OS to be fully controllable by the TS. It controls the application to be tested, and uses its corresponding kernel-level component to control and monitor OS behavior and the OS-level abstractions used by the application, including system calls, virtual memory, file systems, sockets, etc.

The TS Controller and TS Proxy are each divided into user-level and kernel-level components. Additional trusted entities of the security architecture that are not under test, such as network servers, may be hosted in the TS and report their activity directly to the TS Controller.

The modified VMM captures events and accesses system state in the SUT. It monitors and controls the hardware with the Event & Attack Module providing hooks into the virtual CPU and virtual devices, as well as into the new Security Hardware Emulator for new hardware not present in the base CPU. The TS Proxy monitors and controls the applications and OS. Communication of events and data between the SUT and TS occurs asynchronously through a network channel for software events/attacks and through a custom channel within the VMM[1] for hardware events/attacks. When synchronization is necessary, either the application or the entire SUT machine can be frozen to preserve state, while the TS and attack scripts continue to execute. Within the virtual machines, the components communicate through a combination of new system calls (to kernel components), hyper-calls (direct to the VMM), signals, and virtual hardware interrupts.

Table 1 lists various events and attacks exposed by the framework for each layer of the system. The lower two layers show the hardware classified into the base hardware (x86 architecture in our work) and the new emulated security architecture.

Hardware events are monitored through the VMM hooks during execution and are as fine-grained as the execution of a single instruction or hardware operation in the SUT. The VMM freezes the SUT as it communicates each event over the inter-VM channel, allowing the TS to possibly change the result of that operation before it completes. Software events and attacks rely on hooks from the TS Proxy into the OS kernel through its kernel module, and to the testing application using its user-mode component. The TS Proxy can also function as a debugger tool reading the application's memory and accessing its symbol table to map variable and function names to virtual addresses. The application can optionally be instrumented to access its state and events.

3.3 Attack Scripts

Attack Scripts reside on the TS and specify how particular attacks are executed on the SUT. They provide step-by-step instructions for monitoring events and dynamically responding to them in order to successfully launch attacks, or detect that an attack was prevented by the security architecture. The scripts act like a state-machine, acting on hardware and software events which are aggregated by the TS. Scripts can be written to form a library of generic attacks, that can be used to attack any application. Alternatively they can be specific to the behavior of the application being tested, written by the user of the framework. The TS Controller reads and executes these scripts and implements the communication mechanisms and control of the SUT as needed.

Table 2 lists the API which the TS Controller exports to the attack scripts. The first group are commands used to launch and control the execution of the application under test on the SUT. The second group of commands control event handling[2], and the last group provides access to SUT state.

The security properties and attacks considered in the threat model do not need to detail the exact method of penetration, but can just focus on the impact of the attacks on the

[1]The hardware channel is implemented over shared memory between each VM's Event & Attack Module.

[2]The watch list can wait for any of the event types in Table 1. Event parameters and data are either passed to the TS directly or are accessible via pointers with `ACCESS_MEM`.

Table 1: Example Events and Attacks

Layer	Events Monitored	Impact of Attack
Protected Application	API function entry/exit, Library calls, User authentication, Network messages, Other application-specific events.	Read/write application data structures, Trigger application API calls, Intercept/modify network messages, Other application-specific attacks.
OS	Memory access watchpoints, Virtual memory paging, File system access, System calls, Process scheduling, Instruction breakpoints, Device driver access, Network socket access, Interrupt handler invocation, etc.	Read/write virtual memory, Read/write kernel data structures, Read/write file system, Intercept/modify syscall parameters or return values, Read/write suspended process state, Modify process scheduling, Intercept/modify network data, Modify virtual memory translations.
Base Hardware (x86)	Privileged instruction execution, Triggering of page faults and other interrupts, Execution of an instruction pointer.	Read/write general registers, Read/write physical memory, Trigger interrupts, Intercept device I/O (e.g. raw network & disk accesses).
Secure Hardware	Execution of new instructions, Triggering of new faults, Accesses to new registers.	Read/write new registers & state, Read/write protected memory plaintext.

Table 2: TS Controller API for Attack Scripts

Function	Description
`h ← INIT()`	Initialize the Controller and return a handle h to access resources.
`EXECUTE(h,app,params)`	Execute the application *app* on SUT with the given parameters *params*.
`INTERRUPT(h,num)`	Trigger an immediate virtual hardware interrupt number *num* on the SUT.
`BREAKPOINT(h,addr)`	Setup a breakpoint to interrupt the SUT at an address (addr).
`EVENTADD(h,eventType)`	Add the *eventType* to watch-list.
`EVENTDEL(h,eventType)`	Delete the *eventType* from the watch-list.
`event ← WAIT(h)`	Blocking call that *waits* for any event in the watch-list to occur in the SUT. Once an event is triggered, the SUT is paused and the TS continues running the attack script. An application exit in the SUT always causes a return from WAIT().
`event ← WAITFOR(h,eventType)`	Similar to WAIT() but waits for the specified event (or application exit), regardless of the watch-list.
`CONT(h)`	Execution of the SUT is resumed, after an event or interrupt.
`ACCESS_GENREG(h,r/w,buf)` `ACCESS_SPREG(h,r/w,buf)`	Reads/writes (*r/w*) the general registers or SP registers of the SUT to/from *buf*.
`ACCESS_MEM(h,v/p,r/w,addr,sz,buf)` `ACCESS_SPMEM(...)`	Reads/writes (*r/w*) *sz* bytes from virtual or physical memory (*v/p*) of the SUT at address *addr* to/from the buffer *buf*. Can access memory regularly or as an SP secure region (accessing the plaintext of encrypted memory).

SUT's state. This is preferred since (1) new attack penetration methods are frequently discovered after a system is deployed and often are not foreseen by the designer, (2) most real attacks result in or can be modeled by the impact of attacks which we provide in Table 1, and (3) the attack scripts themselves can be restricted to model specific penetration methods when testing for a more limited attacker. A detailed example using this TS Controller API in an attack script is given in Section 5 and Figure 3.

3.4 Implementation

We implemented our testing framework on VMware's virtualization platform [2], including all of the components in Figure 1, and events and attacks at each system layer. The Security HW Emulator, VMM Event & Attack Module, and inter-VM communication channel required modifying the source code of the VMware VMM. The kernel components of the TS Proxy and TS Controller are implemented as Linux kernel modules. The TS Proxy application is implemented as a Linux user process and controls the execution of the Application under test. The TS Controller application is implemented as a static library which is called by the Attack Scripts.

As a sample security architecture, we implement the SP architecture, described in Section 4. The Security Hardware Emulator emulates the SP architecture including its hardware roots of trust, secure memory, and interrupt protection. We have also implemented a library of protected software for SP, which is used for a remote key-management application as described in Section 5. Our Application under test uses this library to exercise the software, and in turn, the SP hardware.

Our framework, by using existing virtualization technology, enables reasonable performance while allowing our SUT to provide a realistic software stack and emulate new hardware. Other virtualization environments, like Xen [4], can also be used. Other simulation and emulation environments available, such as Bochs [28] and Qemu [5], could be used in place of virtualization to implement our framework as designed and described in this paper. We choose a virtualization environment for performance reasons, because only parts of the hardware and protected software need to be em-

ulated, while the OS and other non-protected software can run virtualized. VMware provided an excellent development environment, under the VMAP program.

4. SP ARCHITECTURE AND EMULATION

We use the Secret Protection (SP) architecture [13, 24] to demonstrate the effectiveness of our framework. SP skips software layers in the conventional trust chain by using hardware to directly protect an application without trusting the underlying operating system. SP protects the confidentiality and integrity of cryptographic keys in its persistent storage which in turn protect sensitive user data through encryption and hashing. These security properties provided by SP need to be validated. Furthermore, it is important to write and test many secure software applications for SP in a realistic environment, where a compromised OS can be a powerful source of attacks.

Our testing framework emulates SP's hardware features using modifications to the VMM. While SP hardware primitives have already undergone a detailed security analysis on paper, the framework can test the robustness of the design and its implementation, as well as discover any potential flaws. Additionally, we modify SP's secure memory mechanisms and then show how our framework can be used to demonstrate that these new hardware features are also resilient to attack.

4.1 Secret Protection (SP) Architecture

In the Secret Protection (SP) architecture (See Figure 2), the hardware primarily protects a Trusted Software Module (TSM), which protects the sensitive or confidential data of an application. Hence, a TSM plus hardware SP mechanisms form a minimalist trust chain for the application. Rather than protecting an entire application, only the security-critical parts are made into a TSM, while the rest of the application can remain untrusted. Furthermore the operating system is not trusted; the hardware directly protects the TSM's execution and data.

Protecting the TSM's execution requires ensuring the integrity of its code and the confidentiality and integrity of its intermediate data. Code must be protected from the time it is stored on disk until execution in the processor. Data must be protected any time when the operating system or other software can access it. This includes storage on disk, in main memory, and in general registers when the TSM is interrupted. To provide this protection, SP provides new hardware mechanisms:

Roots of Trust: SP maintains its state using new processor registers; the threat model of SP assumes the processor chip to be the security boundary, safe from physical attacks which are very costly to mount on modern processors. As shown in Figure 2, SP uses two on-chip roots of trust: the Device Root Key and the Storage Root Hash.

Code Integrity: The Device Root Key is used to sign a MAC (a keyed cryptographic hash) of each block of TSM code on disk. When a TSM is executing, the processor enters a protected mode called Concealed Execution Mode (CEM). As the code is loaded into the processor for execution in the protected mode, the processor hardware verifies the MAC before executing each instruction.

Data Protection: For the TSM's intermediate data, while in protected mode, the TSM can designate certain memory accesses as "secure", which will cause the data to be encrypted and hashed before being evicted from on-chip caches to main memory. This secure data is verified and decrypted when it is loaded back into the processor from secure memory. Secure data and code are tracked with tag bits added to the on-chip caches.

Interrupt Protection: Additionally, the SP hardware intercepts all faults and interrupts that occur while in the protected mode before the OS gets control of the processor. SP encrypts the contents of the general registers in place, and keeps a hash of the registers on-chip; When the TSM is resumed, the hash is verified before decryption of the registers.

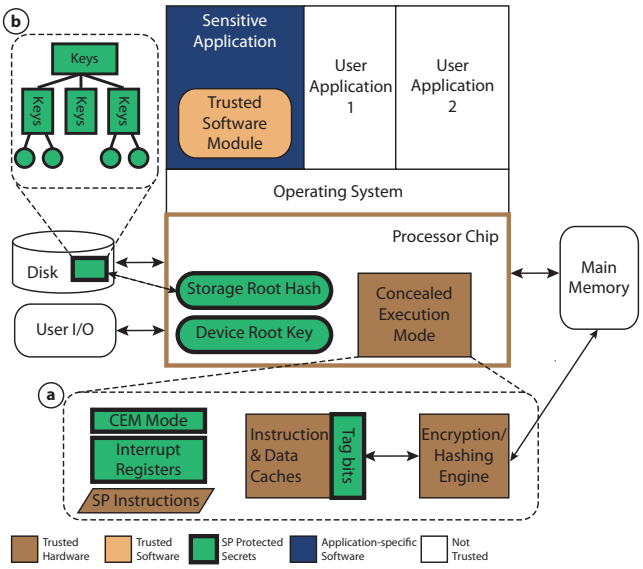


Figure 2: Secret Protection (SP) Architecture. Enlargements show (a) the Concealed Execution Mode (CEM) hardware, and (b) the application secrets protected by the TSM.

The TSM protects secret data belonging to the application in persistent storage. SP allows a TSM (and no other software) to derive new keys from the Device Root Key using a new hardware instruction, *DRK_DeriveKey*. These derived keys are used by the TSM to protect the confidentiality of its persistent data. Furthermore, the TSM is the only software that can read and write the Storage Root Hash register, using it as the root of a hash tree to protect the integrity of this persistent secure data.

Hence, to emulate SP hardware we require the following components: new processor registers (including the protected mode and roots of trust); new instructions; hardware mechanisms for code integrity checking, secure memory and interrupt protection; and new hardware faults which these mechanisms generate.

4.2 Emulation of the SP Architecture

Most of the time, code in a VM runs directly on the physical hardware, and the VMM only emulates components that are virtualized. It traps on privileged instructions, but ignores hardware effects that are transparent to software, such as cache memory. In order to implement and emulate new hardware architecture features, we take advantage

of the VMM's virtualization methods. For example, the VMM maintains data structures for the virtual CPU state, which we expand to store new security registers. The VMM then emulates accesses that are made to those new registers. Other useful VMM behaviors include: interception of all hardware interrupts, dynamic binary translation of code, mapping of virtual memory translations, and virtualization of hardware devices.

To emulate the SP architecture, the Security Hardware Emulator Module implements the following:

Protected Mode: SP requires new registers to be added to the virtual CPU. This includes SP's two Roots of Trust and the new interrupt handling registers and mode bits for its Concealed Execution Mode [24]. New SP instructions are modeled as hypercalls, where the TSM running in the SUT is able to directly invoke the emulation module without going through the guest OS.

Interrupts and SP Faults: The SP architecture changes the hardware interrupt behavior when in protected mode. Since the VMM already emulates interrupt behavior, we simply detect that an interrupt has occurred during the protected mode and emulate the effect on the CPU, which includes suspending the protected mode and encrypting and hashing the general registers. To detect returning from an interrupt, the VMM inserts a breakpoint at the current instruction pointer where the interrupt occurs, so that it is invoked to emulate the return-from-interrupt behavior of SP. Additionally, when the emulated hardware generates a new fault, it first reports to the TS Controller and then translates the fault into a real x86 fault, such as a general protection fault, which is raised in the SUT causing the OS to detect the failure of the TSM.

Secure Memory: We change the SP abstraction of secure memory, as described in Section 4.3. Further, we use block sizes of virtual memory pages rather than individual cache lines, since the VMM does not intercept cache memory accesses. While this limits the ability to model a few low-level attacks on SP (such as the behavior of cache tags), the majority of the security properties of the hardware and all those of the software can still be tested.

Code Integrity: The TSM's code is signed with a keyed hash over each cache-line of code and the virtual address of that line, and is checked as each cache line is loaded into the processor during execution. We model this using the VMM's binary translator to execute the TSM code. Verified instructions are tagged as secure code fragments in the dynamic binary translator cache.

4.3 Lesson Learned from SP Emulation: Secure Memory

The original SP architecture uses two new instructions for a TSM to access secure memory: *Secure Load* and *Secure Store*. With these, any virtual address can be accessed as secure memory, where cache lines are tagged as secure (accessible only to a TSM) and are encrypted and MACed upon eviction from cache. We introduce a new secure memory model, called *Secure Areas*, to replace *Secure Load/Store*.

There are a few drawbacks to the *Secure Load/Store* approach. First, while most new SP instructions can be used as inline-assembly, the compiler must be modified to emit the secure memory instructions whenever accessing protected data structures or the TSM's stack. This further requires programmers to annotate their code to indicate which data structures and variables to protect, and which code and functions are part of a TSM. Second, while a RISC architecture need only supplement a few *Load* and *Store* instructions with their secure counterparts, a CISC architecture has many more instructions which access memory rather than general registers and need to support secure memory access. Third, while SP provides confidentiality and integrity for its secure memory, replay protection is also required to prevent manipulation of the TSM's behavior, but was not explicitly described. Rather, SP assumes a memory integrity tree [38, 14, 9] spanning the entire memory space, requiring significant overhead in on-chip storage and performance when only small amounts of memory need protection.

Secure Areas address these concerns by allowing the TSM to define certain regions of memory which are always treated as secure when accessed by a TSM. The programmer specifies the address range to protect explicitly, allowing the compiler to use regular memory instructions without modification. This is especially useful for our framework since the new architectural features can be tested during design-time without modifying the existing compilation toolchain. It also no longer requires duplicating all instructions in the instruction set which touch memory, a benefit for implementing SP on x86. Finally, it confines the secure memory to a few small regions which are more easily protected from memory replay attacks with less overhead.

Table 3 shows the new instructions added to SP to support Secure Areas, replacing the *Secure Load* and *Secure Store* instructions. The SP hardware offers a limited number of Secure Area regions, which the TSM can define using these instructions. Each region specifies an address range which is always treated as secure memory when accessed by the TSM, and is encrypted when accessed by any other software or hardware devices.

The on-chip secure cache tag bits (shown in Figure 2) are no longer needed; instead $k * 2$ registers are added for defining the start-address and size of k Secure Areas. On-chip storage is also needed to store hashes for each block within the region. The block size for hashing can range from one cache line to one virtual memory page, and is determined by the hardware implementation. Upon defining a new region, the corresponding on-chip hashes are cleared. As secure data is written, it is tagged as secure in cache; when it is evicted from cache, the contents are encrypted and a hash is computed and stored in the on-chip storage for that block. It must be verified when the data is read back in from off-chip memory. Since the regions can be small relative to total memory (only tens to hundreds of kilobytes are needed for our prototype TSMs), only small amounts of on-chip storage are required. Alternatively, other memory integrity tree methods [14, 20, 38] can be integrated to store some hashes off-chip to permit replay protection of larger regions of secure memory.

While both the original SP *Secure Load* and *Secure Store* instructions and the currently proposed Secure Areas have their advantages and disadvantages, the latter is easier to emulate and validate, and requires simpler application software changes.

4.4 Other Architectures

While this paper focused on testing the hardware and software mechanisms of the SP architecture, our testing framework is by no means limited to this architecture. Although

Table 3: New SP Instructions for Secure Areas (only available to TSM)

Instruction	Description
`SecureArea_Add Rs1,Rs2,`*num* Rs1 = start_addr Rs2 = size *(must be aligned to block size)*	Initialize the specified Secure Area (region *num*). On-chip hashes for the region are cleared. All TSM memory accesses for *addr* will be treated as secure if: (start_addr) $\leq$ *addr* $<$ (start_addr + size).
`SecureArea_Relocate Rs1,`*num* Rs1 = start_addr	Change the starting address of the specified Secure Area region. The size remains unchanged. When TSM code in multiple process contexts share memory containing a Secure Area, each may access it at a different address in their virtual address space; this is used to relocate the region.
`SecureArea_Remove` *num*	Disables and clears the specified Secure Area region. On-chip hashes for the region are cleared and secure-tagged cache entries in its address range are invalidated, making any data in the region permanently inaccessible in plaintext.
`SecureArea_CheckAddr Rd,`*num* `SecureArea_CheckSize Rd,`*num*	Retrieves the parameters of the specified Secure Area region. Used to verify whether or not a region is setup for secure memory and where it is located.

other hardware security architectures such as XOM [25], AEGIS [39] and Arc3D [18] have somewhat different goals and assumptions from SP, they combine hardware and software in ways that also make them suitable for validation in our framework. Similarly, TPM [40] adds hardware to protect all software layers and provide cryptographic services. Rather than utilizing changes to the processor itself, TPM adds a separate hardware chip that integrates with the system board. This is still compatible with our testing framework, simply requiring a different set of modifications to the VMM to implement a virtual TPM device. In particular, the ability to observe and control the SUT by use of our components in the framework (TS controller, TS proxy and VMM modifications) can be applied to testing security architectures. Furthermore, software-only security architectures can benefit from analysis under attack in our framework, both during development and for security validation. Access to existing hardware state provides insight into attack impacts and possible flaws, and provides an additional vector for injecting attacks.

5. TESTING OF SP

We now illustrate how we use the Testing Framework to validate a hardware-software architecture like SP, by testing the system's security properties while it is under attack. We also validate that the emulation of the SP mechanisms is correct and secure according to the design, as it forms the basis for the other tests.

Table 4 lists various attacks on the system's security properties. Data confidentiality is the primary purpose of the SP architecture. The attack generally checks to see if any sensitive data that should be protected by a TSM is ever leaked. We eavesdrop on the unprotected memory and check whether any known keys generated by the TSM, in addition to the Device Root Key (DRK) and any DRK-derived keys, are found. This is similar to the cold boot attack [19] which looks for sensitive keys left in physical memory. If the TSM properly uses secure memory for its intermediate data, and protects its persistent data, then no keys should ever leak.

The second section in Table 4 sets up a series of attacks on the basic mechanisms of SP, such as controlling access to the master secrets (e.g., Device Root Key), code integrity checking, and encryption of secure data in protected mode. These tests verify that the emulation is correct and also validate the original security analysis. For example, we attack SP's Concealed Execution Mode by attempting to modify registers during an interrupt. A non-TSM application's registers can be modified by a corrupted OS without detection, causing changes in the application's behavior. However, a TSM will have its registers encrypted and hashed by the SP hardware upon any interrupt, such that SP detects the modification when resuming the TSM.

The next section in Table 4 shows generic attacks on a TSM, which test security properties common to many TSMs (e.g., control flow, entry points). These attacks consider that a basic goal of many TSMs (and indeed of the SP architecture) is to provide confidentiality and integrity to any sensitive information and enforce access control.

We develop tests of the robustness of the TSM against future unknown vulnerabilities that might arise in the hardware or TSM code. Since the penetration mechanism is unknown, we instead model the effects of the attack. For example, the control flow of the TSM could be attacked in many different ways. When the TSM makes branching decisions, the jump targets and the input data for the branch conditions should be protected. If either is not stored in secure memory, or if secure data can be modified or replayed, then arbitrary changes to the TSM's control flow would be possible. We verify that a TSM only bases control flow decisions on data in its secure memory, and test how control flow violations could cause data to leak.

As another example, we consider control flow attacks that allow arbitrary entry points into a TSM. Since instructions to enter protected mode (*Begin_TSM*) are not signed, *Begin_TSM* could be injected into the TSM to create an entry point. We implement this as an attack script, crafting a case where the Testing System overwrites instructions and tries to enter in the middle of a TSM function without detection, bypassing access control checks.[3] To prevent this, we add a new security requirement to SP that it must distinguish entry points in TSM code from blocks of code that are not entry points. This can be achieved by adding an extra bit to the calculation of the signature of each block of TSM code, indicating whether or not it is an entry point.

The attack on TSM page mappings demonstrates a system-

[3]In some cases, this attack would be detected by SP — if the injected instruction is not correctly aligned to the start of a block of signed code, or if later in execution the TSM jumps back to code before the injection site. A carefully crafted attack succeeds.

Table 4: Example Attacks on the SP Architecture Using the Testing Framework

Security Property	Attack
Data Confidentiality	Scan physical memory for leaks of Device Root Key, DRK-derived keys, and TSM's other sensitive information.
Securing General Registers on Interrupts	Attack the general registers during an interrupt of a TSM through eavesdropping, spoofing, splicing, and replay.
Code Integrity	Attack TSM code during execution through spoofing and splicing; attack TSM code on disk.
Secure Memory	Attack intermediate data of TSM through eavesdropping, spoofing, splicing and replay; attack the use of secure memory for TSM's data structures or stack.
Secure Storage	Attack the TSM's secure storage for persistent data (splicing, spoofing & replay).
Control Flow Integrity	Attack TSM's indirect jump targets that are derived from unprotected memory. Arbitrarily modify jump targets within the TSM.
	Attack the input data for branch conditions in the TSM from unprotected memory. Replay secure data to cause incorrect branch decisions.
	Attack TSM entry points by entering CEM at arbitrary points in the code, skipping access control checks or initialization of secure memory.
TSM Page Mappings	Remap TSM code pages and data pages, as a means to attack secure memory or control flow.
Key-chain management	Spoof key add/delete message; replay key-add message after it is deleted; corrupt a key-management message in transit.
Access control on keys	Exceed usage limits/expiration of keys; attempt to use a key that was deleted; attempt to perform a disallowed operation with a key.

level attack. Rather than attacking the TSM directly, the OS manipulates the system behavior to indirectly affect how the TSM executes. The OS can manipulate process scheduling, intercept all I/O operations, and in this case, modify how virtual addresses map to physical addresses.

The last section in Table 4 shows application-specific attacks for a particular TSM — in this case our Remote Key-management TSM. For remote key-management, we consider a trusted authority which owns multiple SP devices and wants to distribute sensitive data to them. The authority installs its remote key-management TSM on each device as well as the protected sensitive data, consisting of secrets and the cryptographic keys that protect those secrets. It also stores policies for each key which dictate how it may be used by the local user. During operation, the TSM will accept signed and encrypted messages from the authority to manage its stored keys, policies, and data. It also provides an interface to the application through which the local user can request access to data according to the policies attached to the keys. The TSM must authenticate the user, check the policy, and then decrypt and display the data as necessary. This TSM stores cryptographic keys, security policies, and secure data in its persistent secure storage, which it protects using SP's underlying hardware mechanisms. We test the confidentiality and integrity of the storage itself, the TSM's use of the storage to protect keys and key-chains, and its enforcement of the policies on accesses to data that the keys protect. We also test the protocols the TSM uses to communicate with a remote authority, managing the keychains.

Our system implements the SP hardware mechanisms, a full TSM providing an API to the application being tested, and a suite of attacks that test both the software and hardware components using our new testing framework. This is a major step towards the complete validation of the design of the SP architecture together with its applications. Furthermore, we demonstrate that TSMs must be carefully written to avoid serious security flaws, and that a security architecture can benefit from testing with many different applications. Our framework provides a platform for this necessary testing, significantly enhancing our ability to reason about the security provided.

Testing Example

Figure 3 shows a sample TSM on the left, and a corresponding attack script using the TS Controller API (Table 2) on the right. This demonstrates the interactions between the TS and SUT for event detection and modification of SUT state. The TSM derives a new key from a nonce it generates and SP's Device Root Key (DRK). It then encrypts a chunk of memory with this new key before sending the encrypted chunk to the network or to storage. The simple attack shown here verifies that secure data (here the derived AES key), placed on the stack by the TSM as a function parameter, is not leaked in physical memory where the OS could read it. This attack is very efficient, assuming a very knowledgeable attacker who is specifically looking for SP derived keys. It demonstrates precise coordination of software events (injected breakpoints) with access to the hardware (physical memory state), while the SUT is frozen to prevent clearing or overwriting of any data in memory. The script also requires access to the internal state of the SP hardware from the TS to verify the results of the attack. Less specific attacks can be constructed, by waiting for any event considered suspicious, then analyzing the event and examining the hardware and software state of the frozen SUT.

Attack scripts are typically longer and can involve many additional steps and interactions, along with a complete TSM and its corresponding application. The full range of events and attack mechanisms in Table 1 are available to the attack scripts, with the TS in full control over the applications, OS, and hardware running in the SUT.

5.1 Lesson Learned: Leaking Data Through the Stack

Application with TSM (`TSMapp`)

```
BEGIN_CEM

...

nonce ← Hash(C_ENC, KeyID)
Reg1 ← DRK_DeriveKey(nonce)
SecureMem.AESkey ← Reg1

// Attack script injects a breakpoint at start of Encrypt function
Ciphertext ← Encrypt(SecureMem.AESkey, &SecureMem.data, sz)
END_CEM

// Send encrypted file on network or store on disk
Network_Send(TTP, Ciphertext, sz)

...
```

Attack Script (pseudocode)

```
EXECUTE(TSMapp, params)

// Wait for key generation
EVENTADD(DRK_DeriveKey)
EVT ← WAIT()
// Read the generated key
ACCESS_SPREG(r, SPRegs)
SPKey ← SPRegs.CEMBuffer

// Inject breakpoint for Encrypt()
BREAKPOINT("&Encrypt"); CONT()
// Wait for interrupt due to breakpoint
EVT ← WAITFOR(Interrupt)

// Scan phys. memory for leaked key
for addr = 0 to 256M − 1 do

  ACCESS_MEM(PHYS, r, addr, 4096, buf)
  if strstr (buf, SPKey) then
    return "Derived Key Leaked in Memory"
return "Derived Key Not Found in Memory"
```

Figure 3: Example Application and Attack Script for Detecting Leaked Keys

In the process of testing how TSMs use SP mechanisms to protect intermediate and persistent data, we found that our new secure memory implementation failed to adequately protect the intermediate data on the stack for a TSM compiled with GCC. Our example TSM in Figure 3 derives a new key from the Device Root Key and uses it for encryption. The attack script freezes the SUT shortly after the key is derived and scans physical memory. It finds that the key has been leaked via parameter passing on the stack, violating data confidentiality. As a result, we have instrumented a new software mechanism to swap the TSM's stack to use memory in a designated Secure Area. The same attack script then verifies that this modified TSM correctly protects the confidentiality of the key when passed as a parameter. This demonstrates how a secure hardware mechanism (e.g., for secure memory) can be used incorrectly by a TSM, often inadvertently, leading to vulnerabilities.

The framework helped significantly in the debugging process, in particular for relocating the TSM's stack to a Secure Area. Even a very simple TSM, which only generates a derived key and saves that key in a secured data structure, manages to leak the key via the stack when using wrapper functions to access new SP instructions. The framework lets us interrupt after critical hardware operations to detect data leaked in plaintext in memory. When we find errors in the way our implementation reassigns stack pointers to use a Secure Area, we can correct the TSM code and the Secure Area setup accordingly, to ensure that all stack operations in a TSM access a valid Secure Area.

Using the framework, we also found a complication when relocating the stack to a Secure Area on an x86 platform. When an interrupt occurs in x86, the processor hardware pushes an exception frame onto the stack, using the stack pointer register; the operating system reads this frame to handle the interrupt. If an interrupt occurs while in CEM, with the stack pointer relocated to a Secure Area, the frame data will be written in the Secure Area region. If still in CEM at the time, this data will be protected as secure memory where the OS will not be able to read it. If CEM has already been suspended before the frame is written, the data will be written in plaintext and will overwrite part of the encrypted and hashed Secure Area data. When CEM is later resumed, the hash check of this region will fail. Therefore, we have developed a new mechanism to make the SP hardware aware of the stack swapping. The hardware saves the original stack pointer in an on-chip register when the stack is relocated. It will automatically restore this original stack pointer before the exception frame is written, saving the secure stack pointer on-chip. The secure stack pointer is then swapped back when CEM is resumed.

The lessons learned are that care must be taken in implementing new trusted software while attempting to use existing software conventions (e.g., for parameter passing through the stack). Also, our testing framework can be used effectively to expose and debug subtle interactions between the trusted and untrusted software and hardware in an implementation.

6. RELATED WORK

One related area of research is the formal verification of both hardware and software, in which mathematical specifications for computer hardware or software are written, and proof techniques are used to determine the validity of such specifications. The complexity of formal verification problems range from NP-hard to undecidable [22, 34, 23, 21]. The complexity of these formal verification mechanisms led to the use of hybrid techniques [7] which use some formal as well as informal methods. Some formal methods of verification include theorem provers (e.g., ACL2 [29], Isabelle/HOL [30]), model checkers [27], and satisfiability solvers [41, 12]. Some informal techniques used in practice are control circuit exploration, directed functional test generation [15], automatic test program generation [11], fuzz testing [17], and heuristic-based traversal [8]. The formal and hybrid techniques try to verify the hardware and software separately, unlike our holistic verification of a software-

hardware system.

The limitation of the formal verification techniques is that they must verify each component piece by piece. This is necessary since the complexity of both specification and verification explodes exponentially with the addition of more pieces to be tested. In our approach, we verify the system in an informal but systematic and efficient way, and consequently we can model both the security critical hardware and software together; we are thus better able to determine the security impacts of the interactions of the various components.

Virtual machine introspection [16, 31, 26] techniques, described previously, provide access to VM-state in similar ways to our framework. However, they focus mostly on observability of software configurations or low-level operating system and hardware behavior. Examples include intrusion detection and virus-scanning from non-vulnerable host systems, preventing execution of malware, and tracing memory or disk accesses. Instead, we strive to combine observability of the full-system state with controllability of those same components, actively during operation, to attack software thought to be secure. In the past work, the focus is on techniques for security monitoring of production machines, rather than design-time testing of new architectures or of new software systems to evaluate their potential vulnerabilities and flaws. Where some of these techniques provide improved hooks into the virtual machine monitor [32], the hooks could be integrated into our framework to make our attack scripts more robust and more flexible.

Chow et al. [10] use system emulation to passively trace data leaks in applications. However, our framework also performs active attacks and looks for violation of security properties. Chow's work also does not consider violations other than data leaks, while we consider more security properties, such as data integrity, policy enforcement, and control flow. Furthermore, we are looking for flaws in trusted code and hardware mechanisms that are specifically designed to protect security, unlike Chow where the applications are tested for properties they were not designed for, therefore leading to unexpected results.

Micro-architectural simulators like Simplescalar [3] are cycle-accurate and hence can be very useful in estimating performance metrics, but they cannot simulate a realistic software system with a full commodity OS. Thus it is impossible to test the security-critical interactions of a software-hardware security solution with such a simulator.

The efforts by IBM [6], Intel [36] and others [37] provide the functionality of a virtual TPM device to software, even when the physical device is not present. In contrast, we not only emulate the new hardware but also hook into the virtual device to observe and control its behavior for testing purposes, and study the interaction with other hardware and software components.

7. CONCLUSION

We have designed and implemented a virtualization-based framework for validation of new security architectures. This framework can realistically model and test a new system during the design phase, and draw useful conclusions about the operation of the new architecture and its software interactions. It also enables testing of various software applications using new security primitives in the hardware or in the OS kernel.

Our framework serves as a rapid functional prototyping vehicle for black-box or white-box testing of security properties. It can utilize and *integrate* multiple event sources and attack mechanisms from the hardware and software layers of the system under test. These mechanisms can test both low-level components and high-level application behavior. As a result, a comprehensive set of attacks are realizable on the hardware, operating system, and applications.

We implement the SP architecture in our framework and test its security mechanisms thoroughly, studying the interactions of trusted software with the hardware protection mechanisms. We also improve the design and implementation of SP's architecture of both the secure memory and the way SP handles dynamic data on the stack. Using a suite of attacks on each layer of the architecture, we thoroughly test each component of SP's trust chain to show the effectiveness of our proposed framework for debugging software, for exposing subtle interactions between existing and new mechanisms and conventions in an implementation, and for reasoning about system security properties.

8. REFERENCES

[1] OKL4 Microkernel. Open Kernel Labs, `http://www.ok-labs.com`.

[2] VMware Workstation. VMware Inc., `http://www.vmware.com`.

[3] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, February 2002.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, 2003.

[5] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[6] S. Berger, R. Caceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: Virtualizing the Trusted Platform Module. In 15^{th} *USENIX Security Symposium*, July 2006.

[7] J. Bhadra, M. S. Abadir, L.-C. Wang, and S. Ray. A Survey of Hybrid Techniques for Functional Verification. *IEEE Design & Test of Computers*, 24(2):112–122, 2007.

[8] G. Cabodi, S. Nocco, and S. Quer. Improving SAT-Based Bounded Model Checking by Means of BDD-Based Approximate Traversals. In *Proc. of the conference on Design, Automation and Test in Europe*, pages 10898–10905, 2003.

[9] D. Champagne, R. Elbaz, and R. B. Lee. The Reduced Address Space (RAS) for Application Memory Authentication. In *Proc. of the 11th International Conference on Information Security*, pages 47–63, 2008.

[10] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *USENIX Security Symposium*, pages 321–336, 2004.

[11] F. Corno, E. Sánchez, M. S. Reorda, and G. Squillero. Automatic Test Program Generation: A Case Study. *IEEE Design and Test of Computers*, 21:102–109,

2004.

[12] G. Dennis, F. S.-H. Chang, and D. Jackson. Modular verification of code with sat. In *Proc. of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*, pages 109–120, 2006.

[13] J. S. Dwoskin and R. B. Lee. Hardware-rooted Trust for Secure Key Management and Transient Trust. In *Proc. of the 14th ACM Conference on Computer and Communications Security*, pages 389–400, October 2007.

[14] R. Elbaz, D. Champagne, R. B. Lee, L. Torres, G. Sassatelli, and P. Guillemin. TEC-Tree: A Low-Cost, Parallelizable Tree for Efficient Defense Against Memory Replay Attacks. In *Proc. of the 9th Cnternational Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 289–302, 2007.

[15] S. Fine and A. Ziv. Coverage directed test generation for functional verification using bayesian networks. In *Proc. of the 40th annual Design Automation Conference*, pages 286–291, 2003.

[16] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[17] P. Godefroid, M. Levin, D. Molnar, et al. Automated whitebox fuzz testing. In *Proc. of the Network and Distributed System Security Symposium*, 2008.

[18] M. Gomathisankaran and A. Tyagi. Architecture Support for 3D Obfuscation. *IEEE Trans. Computers*, 55(5):497–507, 2006.

[19] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *USENIX Security Symposium*, pages 45–60, 2008.

[20] W. E. Hall and C. S. Jutla. Parallelizable Authentication Trees. In *Selected Areas in Cryptography*, pages 95–109, 2005.

[21] R. C. Ho, C. H. Yang, M. Horowitz, and D. L. Dill. Architecture Validation for Processors. In *Proc. of the 22nd annual international symposium on Computer architecture*, pages 404–413, 1995.

[22] W. A. Hunt. Mechanical Mathematical Methods for Microprocessor Verification. In *Intl. Conference on Computer Aided Verification*, pages 523–533, 2004.

[23] H. Iwashita, S. Kowatari, T. Nakata, and F. Hirose. Automatic test program generation for pipelined processors. In *Proc. of the 1994 IEEE/ACM International Conference on Computer-aided design*, pages 580–583, 1994.

[24] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. S. Dwoskin, and Z. Wang. Architecture for Protecting Critical Secrets in Microprocessors. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 2–13, 2005.

[25] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proc. of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 168–177, 2000.

[26] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *the 17th USENIX Security symposium*, pages 243–258, 2008.

[27] A. Lomuscio, H. Qu, and F. Raimondi. MCMAS: A Model Checker for the Verification of Multi-Agent Systems. In *Computer Aided Verification, 21st International Conference*, pages 682–688, 2009.

[28] D. Mihocka and S. Shwartsman. Virtualization Without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure. In 1^{st} *Workshop on Architectural and Microarchitectural Support for Binary Translation in ISCA-35*, June 2008.

[29] S. S. Moore. Symbolic Simulation: An ACL2 Approach. In *Formal Methods in Computer-Aided Design*, pages 334–350, 1998.

[30] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle's Logics: HOL*, 2008.

[31] B. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *IEEE Symposium on Security and Privacy*, pages 233–247, May 2008.

[32] B. Payne, M. de Carbone, and W. Lee. Secure and Flexible Monitoring of Virtual Machines. In *Proc. of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)*, pages 385–397, Dec. 2007.

[33] G. Popek and R. P. Goldberg. Formal Requirements for Virtualizable 3rd Generation Architectures. *Communications of the A.C.M.*, 17(7):412–421, 1974.

[34] S. Ray and W. A. Hunt. Deductive Verification of Pipelined Machines Using First-Order Quantification. In *Intl. Conference on Computer Aided Verification*, pages 31–43, 2004.

[35] M. Rosenblum and T. Garfinkel. Virtual machine monitors: current technology and future trends. *IEEE Computer*, 38(5):39–47, 2005.

[36] V. Scarlata, C. Rozas, M. Wiseman, D. Grawrock, and C. Vishik. *Trusted Computing*, chapter : TPM Virtualization: Building a General Framework, pages 43–56. 2008.

[37] M. Strasser, H. Stamer, and J. Molina. Software-based TPM Emulator. `http://tpm-emulator.berlios.de`.

[38] G. E. Suh, D. Clarke, B. Gassend, M. v. Dijk, and S. Devadas. Efficient Memory Integrity Verification and Encryption for Secure Processors. In *MICRO 36*, page 339, 2003.

[39] G. E. Suh, D. E. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing. In *Intl. Conference on Supercomputing*, pages 160–171, 2003.

[40] Truscted Computing Group. *Trusted Platform Module Specification Version 1.2 Revision 103*, July 2007.

[41] M. N. Velev and R. E. Bryant. Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors. In *Proc. of the 38th annual Design Automation Conference*, pages 226–231, 2001.

Two methodologies for physical penetration testing using social engineering

Trajce Dimkov, André van Cleeff, Wolter Pieters, Pieter Hartel
Distributed and Embedded Security Group
University of Twente, The Netherlands
{trajce.dimkov, a.vancleeff, wolter.pieters, pieter.hartel}@utwente.nl

ABSTRACT

Penetration tests on IT systems are sometimes coupled with physical penetration tests and social engineering. In physical penetration tests where social engineering is allowed, the penetration tester directly interacts with the employees. These interactions are usually based on deception and if not done properly can upset the employees, violate their privacy or damage their trust toward the organization and might lead to law suits and loss of productivity. We propose two methodologies for performing a physical penetration test where the goal is to gain an asset using social engineering. These methodologies aim to reduce the impact of the penetration test on the employees. The methodologies have been validated by a set of penetration tests performed over a period of two years.

Keywords: penetration testing, physical security, methodology, social engineering, research ethics

1. INTRODUCTION

A penetration test can assess both the IT security and the security of the facility where the IT systems are located. If the penetration tester assesses the IT security, the goal is to obtain or modify marked data located deep in the organizations network. Similarly, in testing the physical security of the location where the IT system is located, the goal of the penetration test is to obtain a specific asset, such as a laptop or a document. Physical and digital penetration tests can be complemented with social engineering techniques, where the tester is allowed to use knowledge and help from the employees to mount the attack.

In digital penetration tests the resilience of an employee is measured indirectly, by making phone queries or sending fake mail that lure the employee to disclose secret information. These tests can be designed in an ethical manner [1] and within the legal boundaries [2]. However, measuring the resilience of an employee against social engineering in a physical penetration test is *direct* and *personal*. When the tester enters the facility of the organization and directly interacts with the employees, she either deceives the employee, trying to obtain more information about the goal, or urges the employee to help her, by letting the tester inside a secure area or giving the tester a credential. The absence of any digital medium in the communication with the employees makes the interaction between the penetration tester and the employee intense, especially if the employee is asked to break company policies.

There are three main consequences from personal interaction between the tester and the employee. First, the employee might be stressed by having to choose between helping a colleague and breaking the company policies. Second, the tester might not treat the employee respectfully. Finally, when helping the penetration tester to enter a secure location, the employee loses the trust from the people who reside in the secure location. For example, employees might stop trusting the secretary when they find out she let an intruder into their office. To avoid ethical and legal implications, organizations may avoid physical penetration testing with social engineering, leaving themselves unaware of attacks where the attacker uses non-digital means to attack the system.

This paper tackles the problem how to perform a physical penetration test using social engineering in the most respectful manner, while still getting results that lead to improving the security of the organization. The contribution of this paper is two methodologies for physical penetration tests using social engineering where the goal is to gain possession of a physical asset from the premises of the organization. Both methodologies are designed to reduce the impact of the test on the employees. The methodologies have been validated by performing 14 live penetration tests over the last two years, where students tried to gain possession of marked laptops placed in buildings of two universities in The Netherlands.

The rest of the paper is structured as follows. In section 2 we present related work and in section 3 we set the requirements for the methodologies. Sections 4 and 5 outline the methodologies, section 6 provides an evaluation of the structure of the methodologies and section 7 concludes the paper.

This research is supported by the Sentinels program of the Technology Foundation STW, applied science division of NWO and the technology programme of the Ministry of Economic Affairs under projects number TIT.7628.

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

2. RELATED WORK

In the computer science literature, there are isolated reports of physical penetration tests using social engineering [3, 4]. However, these approaches focus completely on the actions of the penetration tester and do not consider the impact of the test on the employees.

There are a few methodologies for penetration testing. The Open-Source Security Testing Methodology Manual (OSSTMM) [5] provides an extensive list of *what* needs to be checked during a physical penetration test. However, the methodology does not state *how* the testing should be carried out. OSSTMM also does not consider direct interaction between the penetration tester and the employees. Barret [6] provides an audit-based methodology for social engineering using direct interaction between the penetration tester and an employee. Since this is an audit-based methodology, the goal is to test *all* employees. Our methodologies are goal-based and focus on the security of a specific physical asset. Employees are considered as an additional mechanism which can be circumvented to achieve the goal, instead of being the goal. Türpe and Eichler [7] focus on safety precautions while testing production systems. Since a test can harm the production system, it can cause unforseeable damages to the organization. In our work the penetration test of the premises of an organization can be seen as a test of a production system.

In the crime science community, Cornish [8] provides mechanisms how to structure the prosecution of a crime into universal crime scripts and reasons about mechanisms how to prevent the crime. We adopt a similar reporting format to present the results from a penetration test. However, instead of using the crime script to structure multiple attacks, we use the script to identify security mechanisms that continuously fail or succeed in stopping an attack.

In social science research, the Bellman report [9] defines the ethical guidelines for the protection of humans in testing. The first guideline in the report states that all participants should be treated with respect during the test. Finn [10] provides four justifications that need to be satisfied to use deception in research. We use the same justifications to show that our methodology is ethically sound.

3. REQUIREMENTS

A penetration test should satisfy five requirements to be useful for the organization. First, the penetration test needs to be realistic, since it simulates an attack performed by a real adversary. Second, during the test all employees need to be treated with respect [9]. The employees should not be stressed, feel uncomfortable nor be at risk during the penetration test, because they might get disappointed with the organization, become disgruntled or even start legal action. Finally, the penetration test should be repeatable, reliable and reportable [6]. We call these the R* requirements:

Realistic - employees should act normally, as they would in everyday life.

Respectful - the test is done ethically, by respecting the employees and the mutual trust between employees.

Reliable - the penetration test does not cause productivity loss of employees.

Repeatable - the same test can be performed several times and if the environment does not change, the results should be the same.

Reportable - all actions during the test should be logged and the outcome of the test should be in a form that permits a meaningful and actionable documentation of findings and recommendations.

These are conflicting requirements. For example:

1. In a realistic penetration test, it might be necessary to deceive an employee, which is not respectful.

2. In a realistic test, arbitrary employees might be social engineered to achieve the goal, which is unreliable.

3. In a reportable test, all actions of the penetration tester need to be logged, which is unrealistic.

Orchestrating a penetration test is striking the best balance between the conflicting requirements. If the balance is not achieved, the test might either not fully assess the security of the organization or might harm the employees.

We propose two methodologies for conducting a penetration test using social engineering. Both methodologies strike a different balance between the R* requirements, and their usage is for different scenarios. Both methodologies assess the security of an organization by testing how difficult it is to gain possession of a pre-defined asset.

The methodologies can be used to assess the security of the organization, by revealing two types of security weaknesses: errors in implementation of procedural and physical policies by employees and lack of defined security policies from the management. In the first case, the tests should focus on how well the employees follow the security policies of the organization and how effective the existing physical security controls are. In the second case, the primary goal of the tests is to find and exploit gaps in the existing policies rather than in their implementation. For example, a test can focus on how well the credential sharing policy is enforced by employees or can focus on exploiting the absence of a credential sharing policy to obtain the target asset.

In this paper we present the two methodologies which reduce the impact of these tests. The environment-focused (EF) methodology, measures the security of the environment where the asset is located. The methodology is suitable for tests where the custodian (person who controls the asset) is not subject of social engineering and is aware of the execution of the test. One example of such test is evaluating the security of the assets residing in the office of the CEO, but not the awareness of the CEO herself. The custodian-focused (CF) methodology is more general, and includes the asset owner in the scope of the test. In this methodology, the owner is not aware of the test. The CF methodology is more realistic, but it is less reliable and respectful to the employees.

4. ENVIRONMENT-FOCUSED METHOD

First, we define the actors in the environment-focused methodology. Then, we introduce all events that take place during the setup, execution and aftermath of the penetration test. Finally, we validate the methodology by conducting three penetration tests and present some insights from the experience.

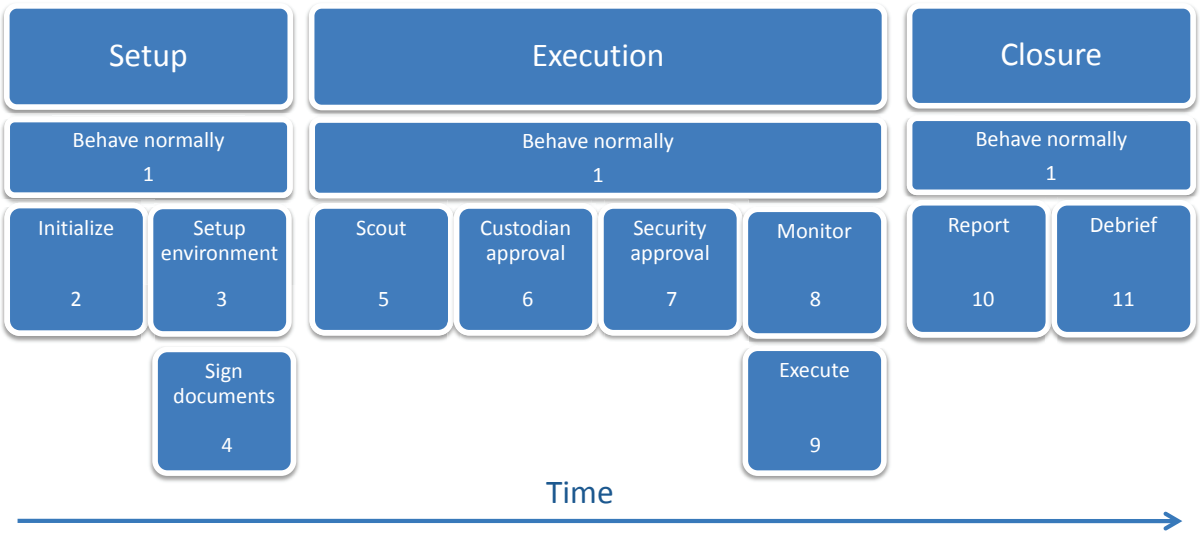


Figure 2: Sequence of events in the environment-focused methodology. Each box represents an event which happens in sequence or parallel with other events. For example, event 3 happens after event 2 and in parallel with events 1 and 4.

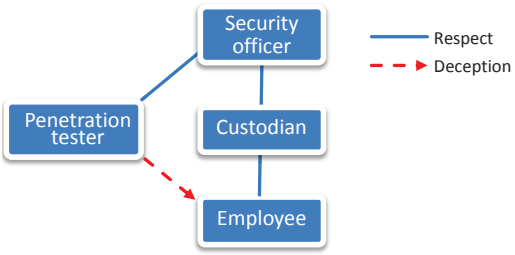


Figure 1: Actors in the EF methodology

4.1 Actors

The penetration test involves four different actors.

Security officer - an employee responsible for the security of the organization. The security officer orchestrates the penetration test.

Custodian - an employee in possession of the assets, sets up and monitors the penetration test.

Penetration tester - an employee or a contractor trying to gain possession of the asset without being caught.

Employee - person in the organization who has none of the roles above.

The actors and the relations between them are shown in Figure 1. The majority of actors treat each other with respect. No respect relation between two actors means either the actors do not interact during the penetration test (for example between the tester and the custodian) or do not have a working relationship (between the penetration tester and the employee). In this methodology, the tester deceives the employee during the penetration test, presented in the figure with a dashed line.

4.2 Setup

Figure 2 provides the sequence of events that take place during the setup, execution and closure of the penetration test. During all three stages of the penetration test, employees should behave normally (1 in Figure 2).

As in other penetration testing methodologies, before the start of the test, the security officer sets the scope, the rules of engagement and the goal (2 in Figure 2). The *goal* is gaining physical possession of a marked asset. The *scope* of the testing provides the penetration tester with a set of locations she is allowed to enter, as well as business processes in the organization she can abuse, such as processes for issuing a new password, or processes for adding/removing an employee. The *rules of engagement* restrict the penetration tester to the tools and means she is allowed to use to reach the target. These rules, for example, define if the tester is allowed to force doors, to break windows or to use social engineering.

The custodian first signs an informed consent form and then sets up the environment, by marking an asset in her possession and installing monitoring equipment.

The asset should not be critical for the daily tasks of the custodian or anyone else, including the organization. Thus, when the penetration tester gains possession of the asset, the productivity of the custodian using the asset and the process flow of the company will not be affected. The custodian leaves the asset in her office or an area without people (storage area, closet). If the custodian shares an office with other employees, the monitoring equipment should be positioned in such a way that it records only the asset and not the nearby employees. The custodian knows when the test takes place, and has sufficient time to remove/obscure all sensitive and private assets in her room and around the marked asset (3 in Figure 2).

Meanwhile, the penetration tester needs to sign the rules of engagement (4 in Figure 2). The OSSTMM methodology [5] provides a comprehensive list of rules of engagement.

4.3 Execution

The security officer should choose a trustworthy penetration tester and monitor her actions during the execution stage.

Generic Script	Attack trace	Circumvented mechanisms	Recommendations
Prepare for the attack	Buy a bolt cutter and hide it in a bag. Scout the building and the office during working hours. Obtain an after working hours access card.	Access control of the building entrances during working hours. Credential sharing policy.	Keep entrance doors to the building locked at all time. Provide an awareness training concerning credential sharing.
Enter the building	Enter the building at 7:30 AM, before working hours. Hide the face from CCTV at the entrance using a hat.	CCTV pre-theft surveillance.	Increase the awareness of the security guards during non-working hours.
Enter the office	Wait for the cleaning lady. Pretend you are an employee who forgot the office key and ask the cleaning lady to open the office for you.	Challenge unknown people to provide ID. Credential sharing policy.	Reward employees for discovering intruders.
Identify and get the asset	Search for the specific laptop. Get the bolt cutter from the bag and cut the Kensington lock. Put the laptop and the bolt cutter in the bag.	Kensington lock.	Get stronger Kensington locks. Use alternative mechanism for protecting the laptop.
Leave the building with the laptop	Leave the building at 8:00, when external doors automatically unlock for employees.	CCTV surveillance. Access control of the building entrances during working hours.	The motion detection of the CCTV cameras needs to be more sensitive .

Figure 3: Reporting a successful attempt. The figure shows an example of a generic script instantiated with an attack trace. First we define the generic script, which encompasses the stages of all attacks. In the example, they are: enter the building, enter the office, identify and get the asset, and exit the building. For each step in a trace, we identify both the mechanisms (if any) that were circumvented and mechanisms that stopped an attack. For failed attacks, the table shows which mechanisms were circumvented up to the failed action, and the mechanism that successfully stopped the attempt.

1. Social engineer night pass from an employee.
2. Enter the building early in the morning.
3. Social engineer the cleaning lady to access the office.
4. Cut any protection on the laptop using a bolt cutter.
5. Leave the building during office hours.

Figure 4: Example of an attack scenario

When the penetration test starts, the tester first scouts the area and proposes a set of attack scenarios (5 in Figure 2). An example of an attack scenario is presented in Figure 4. The proposed attack scenarios need to be approved first by the custodian (6 in Figure 2) and then by the security officer (7 in Figure 2). The custodian is directly involved in the test and can correctly judge the effect of the scenario on her daily tasks and the tasks of her colleagues. The security officer needs to approve the scenarios because she is aware of the general security of the organization and can better predict the far-reaching consequences of the actions of the tester.

If the custodian or the security officer disapprove an attack scenario, they need to evaluate the scenario and estimate the success. The tester puts in the report that the scenario was proposed, the reasons why the scenario was turned down and the opinion of all three roles on the success of the scenario. In this way the scenario although not executed, it is documented including the judgment on the effectiveness of the attack by the security officer, the custodian and the tester.

After approval from the custodian and the security officer, the tester starts with the execution of the attack scenarios (8 in Figure 2). The custodian and the security officer remotely monitor the execution (9 in Figure 2) through CCTV and the monitoring equipment installed by the custodian.

The penetration tester needs to install wearable monitoring equipment to log her actions. The logs serve three purposes. First, they ensure that if an employee is treated with disrespect there is objective evidence. Second, the logs prove that the penetration tester has followed the attack scenarios, and finally, the logs provide information how the mechanisms were circumvented, helping the organization repeat the scenario if needed.

4.4 Closure

After the end of the test, the penetration tester prepares a report containing a list of attack traces. Each attack trace contains information of successful or unsuccessful attacks (10 in Figure 2). Based on the report, the security officer debriefs both the custodians and any deceived employees during the test (11 in Figure 2).

Reporting. The attack traces are structured in a report that emphasizes the weak and the strong security mechanisms encountered during the penetration test, structured following 25 techniques for situational crime prevention [11]. For different domains there are extensive lists of security mechanisms to enforce the 25 techniques (for example, [12]). The combination of the attack traces together with the situational crime prevention techniques gives an overview of the circumvented mechanisms [13] (Figure 3)

Debriefing the employees and the custodian. After finding they were deceived by the same organization they work for, the employees might get disappointed or disgruntled. At the end of the test the security officer fully debriefs the custodian and the employees. The debriefing should be done carefully, to maintain or restore the trust between custodian and the employees who helped the tester to gain the asset.

4.5 Validation

To test the usability of the physical penetration tests using social engineering on the employees, we executed a series of penetration tests following the EF methodology. These pilots allowed us to gain a clear, first-hand picture of each execution stage of the methodology, and draw observations from the experience.

To avoid bias in the execution of the tests, we did not perform the tests ourselves, but recruited three teams of students who were in their first year of master studies to steal three laptops from the custodian (the first author). We locked the laptops with Kensington locks and hid the keys in an office desk. To monitor the laptops, we installed motion detection web cameras which streamed live feeds to an Internet server. Since the custodian shares the office with four other colleagues, the cameras were positioned in such a way to preserve the privacy of the colleagues. We told the colleagues we are doing an experiment, but we did not reveal the nature nor the goal of the experiment.

Since we knew about the penetration test, we did not allow the students to gain possession of the laptops in our presence. During the experiment, we carried on the normal work, thus the students were forced to carry on the attacks after working hours or during the lunch break.

The three teams scouted the building and wrote a list of attack scenarios they want to execute. Eventually, all three teams successfully obtained the target laptop and wrote the successful and unsuccessful attempts in the format shown in Figure 3. After the penetration test, we individually debriefed the security officer, the security guard, the secretary and the colleagues.

4.6 Lessons learned from the penetration tests

The observations are result of our experience with the penetration tests using qualitative social research and might not generalize to other social environments. However, the observations provide an insight of the issues that arose while using the methodology in practice.

The attack scenarios should be flexible. Although the students provided scenarios prior to all attacks, in all cases they were forced to deviate from them, because the target employee was either not present or was not behaving as expected. Attack scenarios assure the custodian and the security officer that the actions of the penetration tester are in the scope of the test, but at the same time there should be some freedom in adapting the script to the circumstances.

The methodology does not respect the trust relationship between the custodian and the employees. After the penetration test, the custodian knows which employees were deceived, and the trust relationship between them is disturbed. For example, if the secretary lets the penetration tester into the office of the custodian, the custodian might not be able to trust her again.

During the penetration test, separating the custodian from the employees is hard. Whenever the students approached a colleague from the office, the first reaction of the colleague was to call the custodian and ask for guidance. This led to uncomfortable situations where we were forced to shut down our phones and ignore e-mails while outside the office.

Debriefing proved to be difficult. After the test, we fully disclosed the test to all involved employees. Debriefing the security guard who opened the office for the penetration testers three times was the hardest. During the debriefing we focused on the benefits of the penetration test to the university and their help setting up the test. After the debriefing, we concluded that we caused more stress to the guard during the debriefing than the students had caused during the penetration test.

5. CUSTODIAN-FOCUSED METHOD

In the EF methodology, the custodian is aware of the penetration test. The knowledge of the penetration test changes her normal behavior and thus influences the results of the test. Since the asset belongs to the custodian, and the asset is in the office of the custodian, in many environments it is desirable to include the custodian's resistance to social engineering as part of the test.

After performing the first series of penetration tests, we revisited and expanded the environment-focused methodology. The CF methodology can be seen as a refinement of the EF methodology, based on the experience from the first set of penetration tests. In the CF methodology the custodian is not aware of the test, making the methodology suitable for penetration tests where the goal is to check the overall security of an area *including* the level of security awareness of the custodian.

5.1 Actors

There are six actors in the CF methodology.

Security officer - an employee responsible for the security of the organization.

Coordinator - an employee or contractor responsible for the experiment and the behavior of the penetration tester. The coordinator orchestrates the whole penetration test.

Penetration tester - an employee or contractor who attempts to gain possession of the asset without being caught.

Contact person - an employee who provides logistic support in the organization and a person to be contacted in case of an emergency.

Custodian - an employee at whose office the asset resides. The custodian should not be aware of the penetration test (1 in Figure 5).

Employee - person in the organization who has none of the roles above. The employee should not be aware of the penetration test (2 in Figure 5).

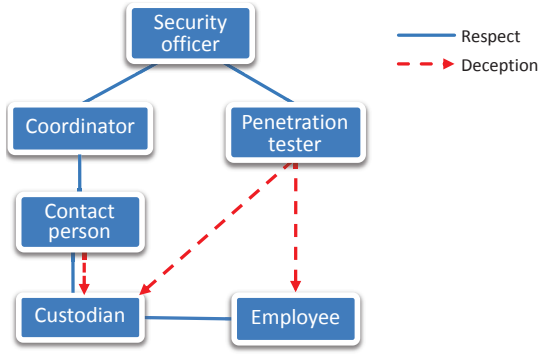


Figure 6: Actors in the CF methodology

Figure 6 shows the actors and the relations between them. In this methodology, the penetration tester deceives both, the employees and the custodian. Moreover, the contact person also needs to deceive the custodian. These relations

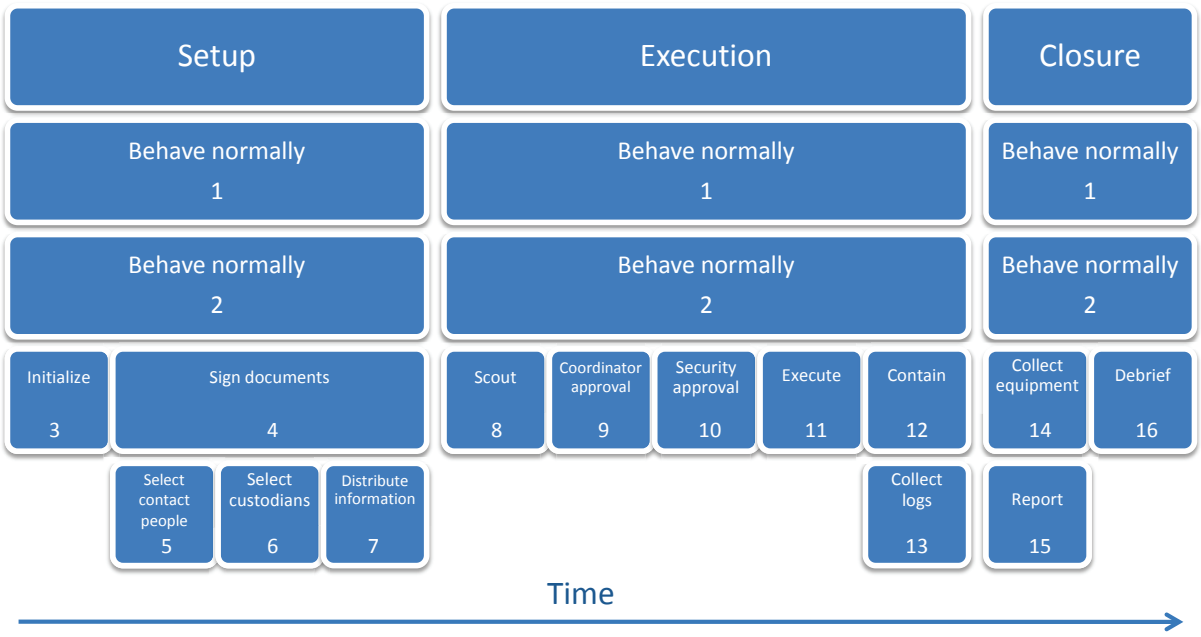


Figure 5: Sequence of events in the custodian-focused methodology

are discussed in greater depth in section 6.

5.2 Setup

At the beginning, similar to the EF methodology, the security officer initializes the test by defining the target, scope and the rules of engagement. The security officer at this point assigns a coordinator for the penetration test and provides the coordinator with marked assets and equipment for monitoring the assets (3 in Figure 5). The marked assets should be similar to the asset of interest for which the security is measured. The monitoring equipment should be non-intrusive and its purpose is to have additional information on the activities of the penetration tester.

The penetration tester should sign the rules of engagement (Appendix A) before the start of the execution stage (4 in Figure 5). The coordinator selects a number of contact people and provides them with the marked assets and the monitoring equipment (5 in Figure 5). Furthermore, the coordinator provides a cover story which explains why the custodian is given the asset. The contact person selects a number of custodians based on the requirements from the security officer (random, specific roles, specific characteristics) and distributes the marked assets and the monitoring equipment to the custodians. After giving the monitoring equipment, the contact person should get a signed informed consent (Appendix B) from the custodians (6 in Figure 5). If the asset can store data, the document must clearly state that the custodian should not store any sensitive nor private data in the asset. Before the penetration test starts, the coordinator distributes a list of penetration testers to the security officer, and a list of asset locations to the penetration tester (7 in Figure 5).

5.3 Execution

The first steps of the execution stage are similar to the previous methodology. The penetration tester scouts the area and proposes attack scenarios (8 in Figure 5). The coordinator and later the security officer should agree with these scenarios before the tester starts executing them (9 and 10 in Figure 5). After approval from both actors, the tester starts executing the attack scenarios. If a penetration tester is caught or a termination condition is reached, the penetration tester immediately informs the contact person. Thus, if the custodian stored sensitive data in the asset, the data is not exposed.

When the tester gains possession of the target asset, she informs the contact person and the coordinator and returns the asset to the contact person (11 in Figure 5). The contact person collects the monitoring equipment and informs the security officer (12 in Figure 5). If the tester gains possession of the asset without the knowledge of the custodian, the contact person needs to reach the custodian before the custodian reaches the office and explain to the custodian that the test is terminated. The security officer obtains surveillance videos from the CCTV and access logs and gives them to the coordinator (13 in Figure 5).

5.4 Closure

After the execution stage, the penetration tester writes a report of all attempts, both failed and successful, in the form of attack traces and gives them to the coordinator (14 in Figure 5). The coordinator has two tasks. First, she collects the marked assets and monitoring equipment from the contact person (15 in Figure 5) and returns them to the security officer. Second, the coordinator debriefs the security officer and the custodians and provides the custodian a form of reward for helping in the assessment (16 in Figure 5).

Not all employees that were social engineered should be debriefed. Employees who were treated with respect and to whom the penetration tester did not cause discomfort during the interaction should not be debriefed, because the debriefing can cause more stress than the interaction with

the penetration tester. The decision which employees need to be debriefed lies with the security officer, and is based on the logs from the penetration tester and the monitoring equipment. The criteria on which employees need to be debriefed are presented in greater detail in Section 6.

Debriefing the custodians. All custodians should be debriefed, because they sign an informed consent at the beginning of the test. However, to preserve the trust between the custodian and the employees, the custodian should not know which employee contributed to the attack.

Three elements should be considered before the debriefing. First, the custodians were deceived by the organization they work for (more specifically, by the contact person). Second, in case of direct interaction, their privacy might be violated by the logging equipment from the tester. Third, they might be stressed from the penetration test either directly, through interaction with the penetration tester, or indirectly, by finding their asset is gone before the contact person reaches them.

The debriefing should focus on the contribution of the custodian in finding the security vulnerabilities in the organization, and the custodian should be rewarded for the participation.

5.5 Validation

We orchestrated eleven penetration tests with the custodian-focused methodology. As targets we used eleven marked laptops that just came out on the market. We selected three PhD students as contact persons, who chose eleven friends as custodians (snowball sampling [14]). The custodians were ten PhD students and one professor spread across two universities, in nine different buildings.

We gave the laptops to the custodians for two weeks usage. The cover story was that the universities are interested in the usability of the new generation of laptops. Thus, we needed to monitor the behavior of the custodian and measure their satisfaction level. The contact persons explained that they cannot tell the custodians exactly which behavior we measure, since it might change the results of the experiment. We instructed the custodians to lock the laptops with a Kensington lock and to keep them in the office. To reduce the risk of data leakage and loss of productivity, we asked the custodians not to store any private or work data on the laptops. Later, thirty-three master students in computer security, divided in eleven teams of three, tried to gain possession of the laptops by using physical and social means.

Figure 7: Recording from the validation of the EF methodology. The student provided to the janitor a fake email stating he needs to collect a laptop from the custodian office. The janitor let the student into the office and helped him find the key from the Kensington lock.

Figure 8: Recording from the validation of the CF methodology. The student went to the office early in the morning, disguised as an employee who forgot his key. The cleaning lady let the student in. The student used a bolt cutter to remove the Kensington lock.

The students took roles as service desk employees, students that urgently needed a laptop for a few hours or coordinator representatives. The students used mobile phones and pocket video cameras to record the conversation with the employees. In one case they took a professional camera and a cameraman, and told the custodian the recording is part of a study to measure the service quality of the service desk.

The resistance of the employees varied. In five cases, the employees gave the laptop easily after being showed a fake email and being promised they will get the laptop back in a few hours. In two cases the custodian wanted a confirmation from a supervisor or the coordinator. In one case a colleague of the custodian got suspicious and sent an email to the campus security. Since only the main security officer knew about the penetration test, in few hours the security guards were all alerted and started searching for suspicious students.

However, in two cases the students were not able to social engineer the custodian directly and were forced to look for alternative approaches. For example, in one of the cases the students entered the building before working hours. At this time the cleaning lady cleans the offices, and under the assumption it is their office let the students inside. After entering the office, the students cut the Kensington lock and left the building before the custodian arrived.

We debriefed only the custodians through a group presentation, where we explained the penetration test and its goal.

5.6 Lessons learned from the validation

It should be specified in advance which information the penetration tester is allowed to use. For example, the penetration tester should not use knowledge about the cover story used by the contact person. During the validation, six penetration testers used knowledge of the cover story to convince the custodian to hand in the laptop. Thus, these tests were less realistic.

Panic situations need to be taken into consideration in the termination conditions. Several times the custodian or an employee got suspicious and raised an alarm. Since only the security officer knew about the experiment, and the other security personnel was excluded, news of people stealing laptops spread in a matter of hours. In these situations the coordinator should react quickly and explain to the employees that the suspicious activity is a test.

The penetration test cannot be repeated many times. If a custodian participated in the penetration test once, she knows what will happen. The same holds for the employees she told about the experiments and the employees that were socially engineered.

6. EVALUATION

In this section we compare both methodologies against the R* requirements. The satisfaction of the requirements is defined by the rules of engagement, which attack scenarios are approved for execution, and the structure of the methodologies. Less restrictive rules of engagement and approving more invasive attack scenarios make the penetration test more realistic, but make the test less reliable and respectful to the employees. The evaluation below assumes these two elements are tuned to the risk appetite of the organization and focuses only on the structure of the methodologies.

Reliable: In the EF methodology, the penetration tester gains possession of a non-critical asset which the custodian is prepared to lose. Thus, the result of the penetration test will not affect the productivity of the custodian. In the CF methodology, the productivity of the custodian may be affected, since the custodian does not know the asset will be stolen. The informed consent is a mechanism to avoid productivity loss, since it explicitly states not to use the marked asset for daily tasks nor store sensitive information on the asset. In both methodologies, the productivity of other employees is not affected, since the penetration tester does not gain possession of any of their belongings without their approval.

Repeatable: The repeatability of any penetration test using social engineering is questionable, since human behavior is unpredictable. Checking if a penetration test is repeatable would require a larger set of tests on a single participant, and a larger number of participants in the test.

Reportable: The approach used in reporting the results of the penetration test completely covers all information needed to perform the attack in a real-life situation and provides an overview of what should be improved to thwart such attempts. The logs from the tester and the monitoring equipment installed by the custodians provide detailed information on all actions taken by the penetration tester, giving a clear overview of how the mechanisms are circumvented.

	EF methodology	CF methodology
Reliable	+++	++
Repeatable	-	-
Reportable	+++	+++
Respectful: actors	++	+
Respectful: trust relations	-	++
Realistic	+	+++

Figure 9: Evaluation of both methodologies

Respectful: Both methodologies should respect all the employees and the trust relationships between them.

In physical penetration testing, the social engineering element is more intense than in digital penetration testing because the interaction between the penetration tester and the employee is direct, without using any digital medium. Baumrind [15] considers deception of subjects in testing as unethical. The National Commission for the Protection of Human Subjects of Biomedical and Behavioral Research, also clearly states this in their first rule of ethical principles: "Respect for persons" [9].

However, some tests cannot be executed without deception. Finn [10] defines four justifications that need to be met do make deception acceptable: (1) The assessment cannot be performed without the use of deception. (2) The knowledge obtained from the assessment has important value. (3) The test involves no more than minimal risk and does not violate the rights and the welfare of the individual. Minimal risk is defined as: "the probability and magnitude of physical or psychological harm that is normally encountered in the daily lives" [16]. (4) Where appropriate, the subjects are provided with relevant information about the assessment after participating in the test. Physical penetration testing using social engineering can never be completely respectful because it is based on deception. However, the deception in both methodologies presented in this paper is justifiable.

The first two justifications are general for penetration testing and its benefits, and have been discussed earlier in the literature (for example, Barrett [6]). The third justification states that the risk induced by the test should be no greater than the risks we face in daily lives. In the EF methodology, the only actor at risk is the employee. The penetration tester cannot physically harm the employee because of the rules of engagement, thus only psychological harm is possible. If the employees help the penetration tester voluntarily, the risk of psychological harm is minimal. The logging equipment assures the interaction can be audited in a case of dispute. In the CF methodology, an additional actor at risk is the custodian. The only case when the risk is above minimal for the custodian is if the tester gains possession of the asset without custodian's knowledge. When the custodian finds the asset missing, her stress level might increase. Therefore it is crucial for the contact person to reach the custodian before custodian learns about the theft.

The fourth justification states that all actors should be debriefed after the exercise. In both methodologies, all actors except the employees are either fully aware of the exercise, or have signed an informed consent and are debriefed after the exercise. Similarly to Finn and Jakobsson [1], we argue that there should be selective debriefing of the employees. Debriefing can make the employee upset and disgruntled and is the only event where the risk is higher then minimal. Thus, an employee should be debriefed only if the security officer constitutes the tester did more than minimal harm.

Besides being respectful toward all the participants, the methodology needs to maintain the trust relations between the employees. The EF methodology affects the trust between the custodian and the employees and the employees and the organization. This is a consequence of the decision to fully debrief all participants in the test. The CF methodology looks at reducing these impacts. First, the custodians are not told who contributed to the attack. Only the coordinator and the security officer have this information, and they are not related to the custodian. Second, the employees are not informed about the penetration test unless it deemed necessary. However, the trust between the custodian and the contact person is shaken. Therefore, the contact person and the custodian should not know each other prior to the test.

In conclusion, the CF methodology is less respectful to the custodian than the EF methodology, because the custodian is deceived and might get stressed when she finds out

the asset is gone. The EF methodology does not preserve any trust between the employees, the organization and the custodian. The CF methodology preserves the trust bond between the custodian and the employees and between the employees and the organization. However, the trust bond between the custodian and the contact person may be affected.

Realistic: The EF methodology allows testing the resilience to social engineering of employees in the organization. Since the custodian knows about the penetration test, she is not directly involved during the execution of the test, making this methodology implementable in limited number of situations. In the CF methodology, neither the custodian nor any of the other employees know about the penetration test, making the test realistic.

One might argue that if the asset is not critical for the employee, the tests are not realistic. On the other hand, taking away "real" assets in the penetration tests will clearly cause loss of production. In the EF methodology, this issue does not exist, as the employees who may be social-engineered are not aware of the importance of the target asset. Therefore, they have no reason to behave differently toward the experimental asset than to a "real" asset. However, in the CF methodology, the value of the asset as perceived by the custodian might influence the result of the tests, as the employee may be more likely to give the asset away if she knows it is not critical. As future work, we plan to investigate the effect of the perceived importance of the asset on the results of such tests.

7. CONCLUSION

Securing an organization requires penetration testing on the IT security, the physical security of the location where the IT systems are situated, as well as evaluating the security awareness of the employees who work with these systems. We presented two methodologies for penetration testing using social engineering. The custodian-focused methodology improves on the environment-focused methodology in many aspects. However, the environment-focused methodology is more reliable, does not deceive the custodian and fully debriefs all actors in the test. We provide criteria to help organizations decide which methodology is more appropriate for their environment. We evaluated both methodologies through analysis of their structure against a set of requirements and through qualitative research methods by performing a number of penetration tests ourselves. This paper shows that physical penetration tests using social engineering *can* reduce the impact on employees in the organization, and provide meaningful and useful information on the security posture of the organization.

In the future, we will focus on two topics. First, we want to investigate the effect of the perceived importance of the asset on the results of the test. We plan to separate the custodians in two groups and inform one of the groups that the laptop contains information critical for the organization. Second, we want to investigate the aspect of *safety* for both the employees and the testers. This research will help penetration testers perform tests in potentially hazardous environment, such as chemical or nuclear laboratories.

References

[1] P. Finn and M. Jakobsson. Designing ethical phishing experiments. *Technology and Society Magazine, IEEE*, 26(1):46–58, Spring 2007.

[2] C. Soghoian. Legal risks for phishing researchers. In *eCrime Researchers Summit, 2008*, pages 1–11. IEEE, 2008.

[3] C. Greenlees. An intruder's tale-[it security]. *Engineering & Technology*, 4(13):55–57, 2009.

[4] Wil Allsopp. *Unauthorised Access: Physical Penetration Testing For IT Security Teams*, chapter Planning your physical penetration test, pages 11–28. Wiley, 2009.

[5] P. Herzog. OSSTMM 2.2–Open Source Security Testing Methodology Manual. *Open source document, www.isecom.org/osstmm*, 2006.

[6] N. Barrett. Penetration testing and social engineering hacking the weakest link. *Information Security Technical Report*, 8(4):56–64, 2003.

[7] S. Türpe and J. Eichler. Testing production systems safely: Common precautions in penetration testing. In *Proceedings of Testing: Academic and Industrial Conference (TAIC PART 2009)*, pages 205–209. IEEE Computer Society, 2009.

[8] D. B. Cornish. The procedural analysis of offending and its relevance for situational prevention. In R. V. Clarke, editor, *Crime Prevention Studies*, volume 3, pages 151–196. Criminal Justice Press, Monsey, NY, 1994.

[9] National Commission for the Protection of Human Subjects of Biomedical and Behavioral Research. The Belmont report: Ethical principles and guidelines for the protection of human subjects of research. pages 1–18, 1978.

[10] P.R. Finn. *Research Ethics: Cases and Materials*, chapter The ethics of deception in research, pages 87–118. Indiana University Press, 1995.

[11] D.B. Cornish and R.V. Clarke. Opportunities, precipitators and criminal decisions: A reply to Wortley's critique of situational crime prevention. *Crime Prevention Studies*, 16:41–96, 2003.

[12] G. Kitteringham. Lost laptops = lost data: Measuring costs, managing threats. Crisp report, ASIS International Foundation, 2008.

[13] R. Willison and M. Siponen. Overcoming the insider: reducing employee computer crime through situational crime prevention. *Communications of the ACM*, 52(9):133–137, 2009.

[14] B.L.A. Goodman. Snowball sampling. *The Annals of Mathematical Statistics*, 32(1):148–170, 1961.

[15] D. Baumrind. Research using intentional deception. Ethical issues revisited. *The American psychologist*, 40(2):165–174, 1985.

[16] Code of Federal Regulations. Title 45: Public welfare department of health and human services. part 46: Protection of human subjects. pages 1–12. 2005.

Appendix A:

Rules of engagement

I, _____ (name of student) agree to perform penetration tests for _____ (name of researcher)

I understand that the participation of is completely voluntary. At any time, I can stop my participation.
I fully oblige to the following rules of engagement:

1. I will only execute attacks that are pre-approved by the researcher and only to an assigned target.
2. I am not allowed to cause any physical damage to university property, except for Kensington locks.
3. I am not allowed to physically harm any person as part of the test.
4. I will video or audio record all my activities while interacting with people during the penetration test as a proof that no excessive stress or panic is caused to anyone.
5. If I am caught by a guard of a police officer, I will not show any physical resistance.

Signature of researcher:_____ Date: _____

Signature of student: _____ Date: _____

Appendix B:

Informed consent

I, _____ (name of employee) agree to participate in the study performed by _____ (name of the research group).
I understand that the participation of the study is completely voluntary. At any time, I can stop my participation and obtain the data gathered from the study, have it removed from the database or have it destroyed.
The following points have been explained to me:

1. The goal of this study is to gather information of laptop usage. Participation in this study will yield more information concerning the habits people have in using mobile devices.
2. I shall be asked to work for 5 min every day on a laptop for one month. The laptop will be monitored and recorded using a keynoter and a web-camera. At the end of the study, the researcher will explain the purpose of the study.
3. No stress or discomfort should result from participation in this study.
4. The data obtains from this study will be processed anonymously and can therefore not be made public in an individually identifiable manner.
5. The researcher will answer all further questions on this study, now or during the cause of the study.

Signature of researcher:_____Date:_____

Signature of employee:_____ Date:_____