Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting

Sherri Sparks, Shawn Embleton, Ryan Cunningham, Cliff Zou
University of Central Florida
{ssparks, embleton, czou}@cs.ucf.edu

Abstract

We present an extension of traditional "black box" fuzz testing using a genetic algorithm based upon a Dynamic Markov Model fitness heuristic. This heuristic allows us to "intelligently" guide input selection based upon feedback concerning the "success" of past inputs that have been tried. Unlike many software testing tools, our implementation is strictly based upon binary code and does not require that source code be available. Our evaluation on a Windows server program shows that this approach is superior to random black box fuzzing for increasing code coverage and depth of penetration into program control flow logic. As a result, the technique may be beneficial to the development of future automated vulnerability analysis tools.

1. Introduction

As the number of households and businesses owning personal computers continues to climb, data and software security are becoming growing concerns. According to the National Vulnerability Database, the number of reported software vulnerabilities has risen from 25 in 1995 to nearly 5000 in 2005 [13]. As a result, there has been a great deal of commercial and academic interest in developing automated software security tools.

Vulnerability analysis involves discovering a subset of the input space with which a malicious user can exploit logic errors in an application to drive it into an insecure state. As software becomes larger and more complex, exploring a commercial application's entire state space for exploitable vulnerabilities becomes an intractable problem. To reduce the scope of exploration, security researchers have developed a number of testing techniques. White box testing, also known as *structural* or *glass box* analysis, typically involves detailed, manual analysis of either program source code or a static disassembly. It is based upon the assumption that the tester has internal knowledge of the system during the test case generation process. In contrast, the black box or *functional* testing methodology views a program as a "black box". It does not rely upon either source code or disassembly. Rather, it is based upon injecting random or semi-random external input into a program and then monitoring its output for unexpected behavior. This process is also sometimes referred to as *fuzz testing* or *fault injection* [7]. Time and cost are motivating factors in application security. Black box *fuzzers* have become popular in recent years because they provide a favorable cost / benefit ratio due to their simplicity and potential for automation.

What black box fuzzers lack, however, is input selection based upon guided feedback concerning progress within the program logic being tested. In practice, security researchers frequently encounter situations where they have analyzed and located a potentially exploitable location in a program which is dependent upon some user controlled input. An example might be a packet received over a network connection by an application that is subsequently sent into some API function known to be vulnerable to buffer overflows. *Exploitability*, however, also implies *reachability*. That is, in order to determine if the vulnerability is an exploitable threat, one must prove that it is reachable on the execution path given some user supplied input. The exact format of this input is dependent upon the control flow logic on the path between the packet acceptance and the basic block where the vulnerable API function is used. Figure 1 provides a graphical illustration of this idea.

Ideally, a fuzzer should have some basic "intelligence" that allows it to preferentially drive execution through any input dependent parse logic to a suspected vulnerable location. We feel that a genetic algorithm is well suited to this task. This was motivated by several observations:

- The runtime execution trace of a program is dependent upon both its user supplied input and the static structural characteristics of its control flow graph. Therefore, if a given input makes it closer to a region of the control flow graph that we wish to

This research was supported by NSF Grant CNS-0627318 and Intel research funds.

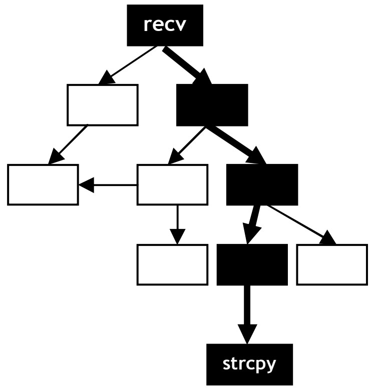

Figure 1. An idealized diagram of the input crafting problem (i.e. what input will cause the program to exercise the control flow logic on the path from the recv function to a potentially vulnerable strcpy()?)

explore than some other input, it may have some desirable characteristics which make it capable of satisfying some (if not all) of the logical constraints on that path. Thus some "invalid" inputs may be better than other "invalid" inputs.

- If we somehow combine or "breed" the best of the invalid inputs we've found in the past, we may select for those characteristics in the future which maximize the number of constraints we are able to satisfy on the execution path and thus increase our overall exploration of the program state space. This is the "survival of the fittest" axiom.

The essential contribution of this paper is threefold. First, we extend traditional "black box" fuzz testing using a genetic algorithm based upon a Dynamic Markov Model fitness heuristic. This heuristic allows us to "intelligently" guide input selection using a genetic algorithm based upon dynamic feedback concerning the "success" of past inputs that have been tried.

Second, we build upon the idea of using a partially specified input structure. The input structure is specified via a context-free grammar and "evolved" using grammatical evolution [16].

Lastly, we focus upon implementing a practical, prototype using existing tools and technologies. We used the open source PAIMEI reverse engineering framework to construct a source code independent testing tool [14]. Thus, our prototype can be easily extended and implemented by others. Our intelligent fuzz testing tool provides focused code coverage and targeted execution control by driving program execution to selective regions of interest in the code (which are suspected to contain vulnerabilities or bugs). Unlike many existing tools, our implementation doesn't require source code. Thus our prototype can be readily used to analyze commodity software. Finally, a grammar frees us from being bound to a particular protocol. If the user wishes to fuzz a new protocol, all he or she has to do is replace the grammar file.

The paper organization is summarized as follows: Section 2 discusses our methodology with implementation specifics covered in Section 3. Experimental evaluations are discussed in section 4. Section 5 follows with an analysis of the limitations of our technique. Related work is discussed in section 6. Finally, in section 7 we conclude with a few ideas for future work.

2. Methodology

In this section, we describe an intelligent black box fuzz testing methodology capable of crafting inputs which force an application to execute specific dependent portions of its control flow graph (see Figure 1). The ability to selectively drive exploration of an application's state space is useful for a fuzz testing tool when the vulnerability analyst wishes to focus testing on a specific portion of a program's control flow graph or "drill down" to a specific node suspected of containing a vulnerability. Clearly a binary response indicating whether a given input reaches the destination state or not is insufficient. We require a smoothed projection of the search space, where some inputs are more nearly correct than others. We use the control flow graph of an application to create such a fuzzy search space.

2.1. Modeling Dynamic Control Flow as a Markov Process

A control flow graph for an executable program is a directed graph with nodes corresponding to blocks of sequential instructions and edges corresponding to non-sequential instructions that join basic blocks (i.e. conditional branch instructions.) A control flow graph for a binary executable can be obtained using a disassembly engine, such as IDA Pro [9].

If we treat the transition behavior of an arbitrary input at a particular basic block in the control flow graph as an estimated parameter, we can develop a probabilistic model called an absorbing Markov process for input behavior from the control flow graph. A Markov process is a type of stochastic process in which the outcome of a given trial depends only on the current state of the process [15]. A system consisting of a series of Markov events is called a Markov chain. If certain outcomes in the Markov chain loop directly back to any state with absolute certainty, it is called an absorbing Markov chain [15]. The edge probabilities for our Markov process correspond to the probability a random input will take a given state transition in the control flow graph. As we are only interested in estimating the probabilities along paths which lead to a desired state, we treat all transitions leading off of the subgraph consisting of these paths as transitioning to a common absorbing state, which we call the *rejection state*. Any program behavior made after reaching the target state can also be ignored, so we treat this node in the control flow graph as an absorbing state as well and call it the *acceptance state*. Figure 2 illustrates this idea.

Rather than determining the absolute probability values for these transitions, we try to estimate them by considering each input tested as a biased statistical sample of this Markov model. The solution space is therefore simply the probability of an input taking a particular execution path in the sampled Markov process. The estimated probability of following a given execution path is therefore:

$$p_i = \frac{tr_i}{bb_i}$$

Where tr_i is the total number of inputs that have taken edge transition i, and bb_i is the total number of inputs which have reached the block containing the conditional statement for edge transition i.

2.2. Searching the Input Space with a Genetic Algorithm

Evolutionary computation is a field of machine learning which attempts to mimic the process of evolution to derive solutions for a certain task [3]. Genetic algorithms are evolutionary algorithms that function as stochastic global optimizers. A *genome* in a genetic algorithm represents one potential solution (e.g. for a problem requiring a string input, the string would be a genome). Genetic algorithms operate iteratively on populations of genomes. Each iteration is called a generation. In each generation, all of the genomes in the population are evaluated by a *fitness function*. The fitness function measures their suitability within their environment. The genomes which score the highest fitness are then selected for crossover. Crossover can be likened to biological breeding. It is a mechanism that allows useful genomes to be combined to produce newer and hopefully more fit genomes. Less fit genomes are simply discarded. This mimics the process of natural selection. Finally, to widely explore the solution space, certain genomes in the population are mutated. Mutation makes a random change within a genome (e.g. bit flip, swapping of bytes, ect).

The genomes in our genetic algorithm implementation correspond to individual inputs for the application. We build each input string from a variable length integer genome using a user-specified context-free grammar and grammatical evolution [2]. The fitness of each genome is the inverse of its execution path's estimated probability derived from the estimated Markov process probabilities on the control flow graph.

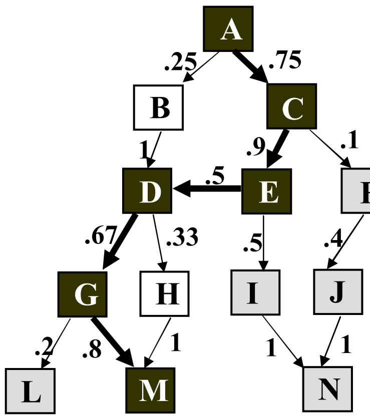

Figure 2. Markov probabilities associated with state transitions on a control flow graph. The white and black squares represent nodes on a control flow path from a given source node (A) to a destination node (M). The grey squares represent reject nodes (nodes from which it is no longer possible to reach the destination node (M)). The black squares represent the path taken by an arbitrary input through the control flow logic. This path consists of node transitions A→C→E→D→G→M. We can calculate the fitness of this input by multiplying the edge transition probabilities:

Fitness = 1 / (.75 × .9 × .5 × .67 × .8) = 5.525

$$\text{Fitness }(x) = \frac{1}{\prod_{l} p_i}$$

Where l is the length of the execution path for input x on the control flow graph, and p_i corresponds to the estimated probability of taking edge i in the control flow graph. Because our fitness function depends on the behavior of all of the other genomes in a generation, it is necessary to compute the fitness value for a particular generation of genomes, we first find the execution path for all members of the population and update the transition probabilities of the Markov chain. We then find the probability of each genome's execution path using the updated dynamic Markov model.

Because the genetic algorithm is attempting to create strings that will follow the least probable execution paths, it will bias our sampling. While this may at first seem to be a disadvantage, it is actually an advantage. The resulting fitness function calculation is actually the probability the genetic algorithm has produced a string which followed a given execution path. This will reward those genomes in the population that represent input that that takes previously unexplored execution paths as well as rare (i.e. difficult) execution paths.

2.3. Grammatical Evolution

Grammatical evolution is a special type of genetic algorithm that can evolve strings in an arbitrary context-free language [2]. Rather than directly encoding the resulting string in its genome, the genome encodes production rules to produce a string from a specified context-free grammar. Each genome is a variable length integer string. The algorithm used to produce a string from the grammar using a genome is summarized in Figure 3. Figure 4 illustrates the construction of a genome from a context-free grammar via grammatical evolution.

```
while ( nonterminals in the string )
{
     find first nonterminal
     numRules = number of production
                rules for the
                nonterminal
     i = next integer in the genome %
         numRules
     apply production rule i
}
```

Figure 3. Pseudocode for grammatical evolution.

Because the input space for the program represents the search space for our program, its combinatorial nature makes blindly generating variable length binary input in a linear genome (as is usual in genetic algorithms) inefficient. Rather than using the usual representation, our genomes represent instructions for building the input from a user-specified context-free grammar, following the Grammatical Evolution paradigm. This not only gives the user the ability to narrow the search space based on knowledge of their specific application, it also gives the genetic algorithm flexibility to represent input with similar structural characteristics as being more nearly adjacent in the search space (e.g. Creating matching HTML tags in a variable
length linear genome would require four insertion mutations for each bracket character and one mutation for the slash character that all happen to occur in the right position, but with the correct grammar, grammatical evolution would require just one insertion mutation that added the tag generation production rule to the genome).

		0		1		2
S	→	s**A**s	|	x**B**x	|	m
A	→	b**B**b	|	**B**		
B	→	a**A**a	|	**C**	|	**AB**
C	→	**C**	|	d	|	e

$$\mathbf{S} \xrightarrow{1} x\mathbf{B}x \xrightarrow{0} xa\mathbf{A}ax \xrightarrow{0} xab\mathbf{B}bax$$
$$\xrightarrow{1} xab\mathbf{C}bax \xrightarrow{1} xabdbax$$

Figure 4. Construction of a genome from a context-free grammar. Construction begins with the initial rule S. The application of production rule S[1] results in the new string xBx. The nonterminal B is then replaced by B[0] to produce the string xaAax. Application of production rules continues in the order of A[0], B[1], C[1] until there are no nonterminals remaining in the string. The final genome consists of the sequence of rule applications {S, B[0], A[0], B[1], C[1]}. Thus, the genome is a sequence of rule applications that forms a "set of instructions" for how an input string should be built.

3. Implementation

In order to implement our approach as a practical tool, we needed to address several requirements:

Disassembly And Control Flow Graph Extraction: The control flow graph used in our methodology is actually a subgraph of the overall program control flow graph. It consists of all basic blocks on a path between an input block and a desired acceptance (i.e. destination) block. All edges leading to blocks off this subgraph are assigned to a special rejection set.
Custom Debugger: We use a debugger for lightweight basic block level binary instrumentation. This is necessary for us to track the runtime execution path and gather the statistics necessary for the genetic algorithm to rate the fitness of individual inputs. Specifically, we set breakpoints in the test application on the entry points of all nodes in the control flow subgraph and rejection set. We then run the test application successively on a randomly supplied population of inputs. In the breakpoint handler, we track the execution path up to the point where the execution path reaches a rejection node (i.e. the destination is no longer reachable along all subsequent execution paths) or terminates in success. At this point, we calculate the fitness for the input probabilistically using the previously discussed Markov Chain heuristic. The fittest individuals are

mated to form the next generation of test inputs. This process of input injection, execution path tracking, fitness evaluation, and mating continues until either a maximum number of generations are reached or the application has been forced into the acceptance state. Figure 5 provides pseudocode for our algorithm.

GA - We implemented the genetic algorithm in Python. The genomes are variable length integer strings, which we convert into input strings using grammatical evolution. We use single point crossover; elitism; and insertion, deletion, and point mutations. If progress in the control flow graph stagnates, the mutation rate dynamically increases until further progress is made.

We chose to use the PAIMEI framework because it provided most of the basic components we needed (e.g. scriptable debugging and support for control flow graph extraction) [14]. PAIMEI is written in Python and exposes functionality that includes a debugger, a graph based binary abstraction, and a set of utilities for accomplishing repetitive tasks. Python was also an ideal language for implementing the genetic algorithm, since it provides extensive built-in support for string manipulation. PAIMEI has been used by software security researchers for a wide range of static and dynamic code analysis tasks like fuzzing, code coverage, and data flow tracking.

4. Evaluation

In the following sections, we evaluate of our tool's potential as a fuzzer. For all of our tests, we tested our tool on the tftpd.exe Windows server program for 50 runs and compared it to a random exploration [17]. We ran the GA for 3,000 generations and the random exploration for equivalent to 10,000 generations worth of input data. Each generation consisted of 50 different test inputs. The context-free grammar we used to generate inputs consisted of hex bytes from 0 to 255 in addition to the mode strings "netascii", "octet" and "mail". We included these strings because we knew them to be part of the tftp packet format and they appeared in the disassembly listing of program strings. Indeed, such strings are easily extracted from a binary program and could potentially provide a rich source for the automatic derivation of application specific grammar rules (clearly the GA will learn faster if it has more useful information in the grammar to generate its test inputs from). The GA parameters we used in our experiments were: Mutation Rate = 90%, Crossover Rate = 75%, Elitism, Selective Breeding, and Dynamic Mutation.

Lastly, all of the following experiments were performed upon a desktop PC containing an Intel Core Duo 2 processor, 1 GB of memory, and the Windows XP Professional operating system. Running times varied, depending upon the exact experiment and whether we were using the GA or the random search. None of the individual runs, however, took longer than a few hours to complete.

4.1. Targeted Execution

Security researchers frequently encounter situations where they have analyzed and located a potentially exploitable location in a program that is dependent upon some user controlled input. An example might be a packet received over a network connection by an application that is subsequently sent into some API function known to be vulnerable to buffer overflows. Exploitability, however, also implies reachability. That is, in order to determine if the vulnerability is an exploitable threat, one must prove that it is reachable on the execution path given some user supplied input. The exact format of this input is dependent upon the control flow logic on the path between the packet acceptance and the basic block where the vulnerable API function is used.

In this section, we present an evaluation of our tool's feasibility for the previously discussed scenario of determining the input structure needed to drive program execution into a potentially vulnerable state. We targeted two published vulnerabilities in the TFTPD server program [17]. These vulnerabilities exist in the packet parsing logic and are the result of improper bounds checking on strings passed into two strcpy() functions.

Our findings were twofold. First, we showed that our genetic algorithm is capable of evolving tftp packets that successfully reach the basic blocks containing the vulnerable strcpy() functions in the control flow graph. Tftpd.exe performs several checks and validations on a packet before it reaches these functions. This demonstrates that our algorithm is capable of learning to navigate the program's internal logic to reach these locations. Second, we demonstrated the superiority of our approach to *fuzzing* with random input. Figure 7 shows a control flow graph generated by our tool during one of our test runs. Figure 6 shows a comparison of the performance of the genetic algorithm to a random search. The GA outperforms the random fuzzing approach. The first vulnerable strcpy() function was reached by the GA in 224 generations on average compared to a random search which took an equivalent of 2294 generations on average. The GA's superiority was more pronounced on the second strcpy() function situated deeper in the logic structure. The GA reached it in an average 227 generations compared to 9,106 for the random brute force.

4.2. Code Coverage Selectivity

While security researchers may want to target a specific vulnerability to discover whether or not it is exploitable, they also investigate the overall behavior of an application. They do this by trying to cover as much of its code as possible. In other words, they attempt to exercise as many different execution paths as possible. This can reveal previously undiscovered bugs, or it can increase the developer's confidence that the application is robust.

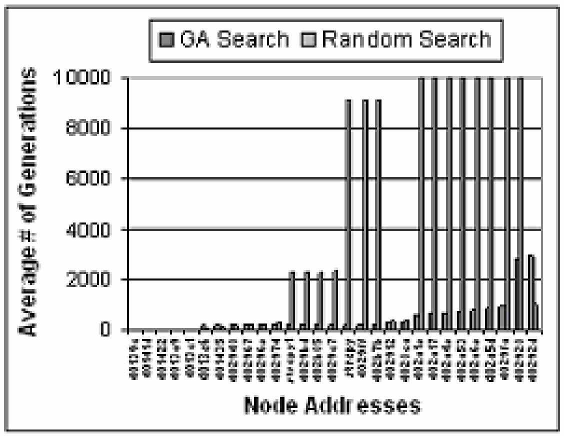

Figure 6: Comparison between GA driven and random search of tftp packet parsing logic. The node address corresponds to basic block addresses on paths from the beginning to the end of the packet parsing logic.

Many current code coverage tools are inadequate and inefficient for vulnerability analysis. Many of these tools focus upon achieving global coverage of the application state space (i.e. reaching all nodes in its control flow graph). Because all but the most trivial applications have extremely large state spaces, achieving such extensive coverage can be time consuming and resource intensive. This may be justifiable for a software tester attempting to ensure the correctness or robustness of a mission critical application. However, software security researchers may wish to focus their resources on the specific subsets of the state space most likely to be targeted by malicious users (e.g. a security analyst should examine parsing logic for a packet received off of an open port more closely than GUI code accepting mouse input.). Because our fitness function drives execution toward less explored regions within a subset of the overall control flow graph, code coverage is a natural extension. Rather than trying to achieve coverage of the entire state space, our tool focuses upon achieving coverage within localized program regions. This is a second area we evaluated our methodology.

During our investigation of the tftpd server's strcpy() vulnerabilities, we collected code coverage data for the program region corresponding to the protocol parsing logic. We compared the coverage obtained with input selection driven by our genetic algorithm against the coverage obtained by the random selection process. Averaged over 50 runs, the genetic algorithm achieved 84.81% coverage of the region compared to 49.54% coverage achieved during the same number of inputs using the random selection. Furthermore, coverage was not appreciably improved by continuing the random selection process for another 7000 generations worth of input data. It slightly increased to 54.51%. We suspected that the natural tendency of our fitness function to drive exploration toward less explored regions of the graph would improve how quickly we were able to achieve specific penetration depths in the control logic structure. The usage of elitism by our GA also ensured that we would never lose the information learned during previous generations.

In another experiment, we assigned a penetration depth value to each node corresponding to the minimum number of edges between it and the source node. We then tabulated the average number of generations required to reach each depth in the control flow graph between the beginning and end of the tftp protocol parsing logic. Figure 8 shows the results of this experiment. As you can see, the GA once again outperforms a random search. Its behavior is also more consistent. We can also observe that nodes at greater depths (> 10 edges) become increasing difficult for the both the GA and the random search to reach. The increase in difficulty is, however, much more linear for the genetic algorithm. No doubt, this is a result of the natural tendency for the GA to leverage what it has learned from the previous observation of dynamic execution response to the inputs it has tried. Finally, we note that although we ran this experiment for input equivalent to 10,000 generations consisting of 50 trial inputs apiece, the random search never hit any of the nodes deeper than 13 edges away from the source. Thus, we should view the data points for these depths as setting a lower bound on the search performance. We expect that that performance will continue to exponentially increase with greater depths.

4.3. Automatic Input Format Learning

Because many programs erroneously trust that user input will conform to well behaved published or implied standards, protocol parsing bugs abound. Protocol parsing code, furthermore, typically has a rich and deep control flow structure. This makes it ideal for testing our genetic algorithm's ability to explore the boundaries imposed by parsing logic on a program's input space. Once again, we applied it to the tftpd server program. By setting the destination node to the basic block indicating acceptance of a valid read or

```
1. Extract program control flow graph using pida_dump.py (provided in PAIMEI
   framework
2. Extract subgraph (source, destination) and set of reject nodes
3. Load program & attach debugger
4  Register breakpoint handler
5. Set breakpoints on subgraph and reject node basic blocks
6. Register exception handler
7. Initialize GA parameters
8. Initialize random population
8. Inject input
9. While destination node not yet reached:
       a. When a breakpointed node is hit:
              i.    Update code coverage information if this is the first time we've visited
                    this node
              ii.  Update the visit count for this node
              iii. Add this node to the path taken by the current input
              iv.  If (the node is a reject state)
                      a. If (we have not yet ran all inputs in the population for the current
                          generation)
                              i.  Inject next input
                              ii. Return
                      b. Else we have ran all inputs in the population for the current
                          generation:
                              i.   Calculate fitness
                              ii.  Build new population via crossover and mutation
                              iii. Inject new input
                              iv.  Return
              v.   Else node is not a reject state:
                      i. Return
```
Figure 5. Pseudocode for our prototype fuzz testing tool

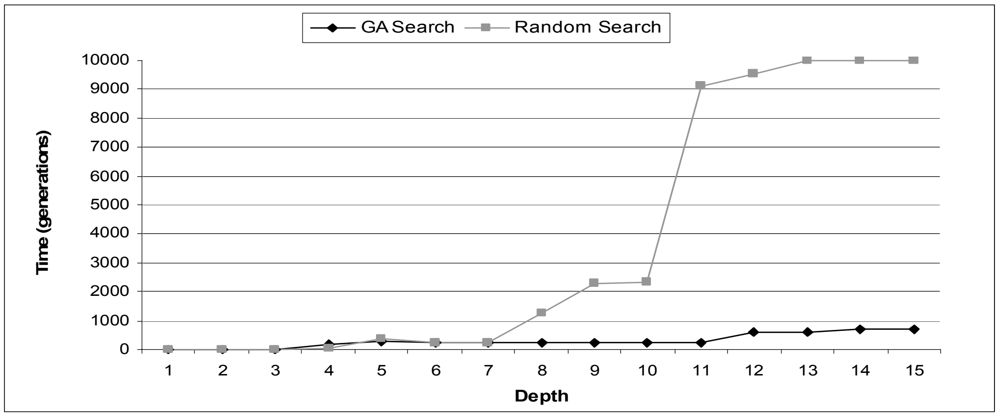

Figure 8. Average # of Generations for CFG Penetration Depth

		2 bytes	string	1 byte	string	1 byte
	Generation	**Opcode**	**Filename**	**0**	**Mode**	**0**
1.	0	2326				
2.	1	626F637465741B7B6225				
3.	51	0005	367D			
4.	72	0002	36060628791E32			
5.	78	0002	36060128	00	0A2A3606	
6.	111	0001	NULL	00	0A057C0561	
7.	393	0002	187566	00	266F6374657464	00
8.	547	0001	2E027D1C02006F63746574	00	6E6574617363696964	00

Figure 9. Evolution of a TFTP Packet

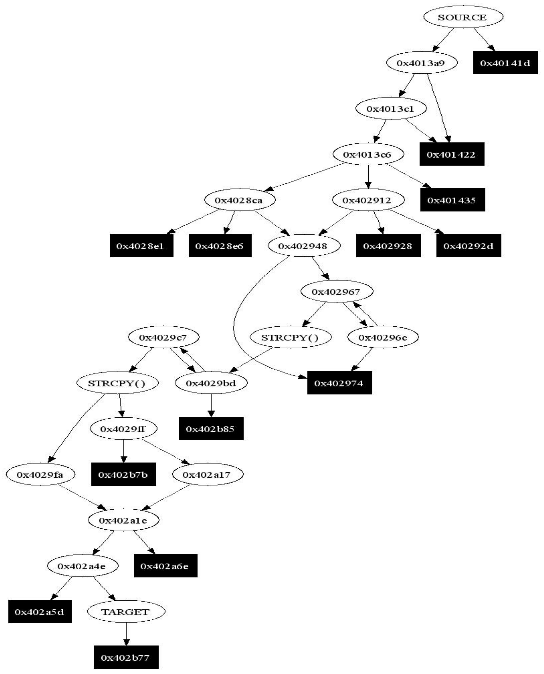

Figure 7. Graph generated by our research tool during exploration of the tftpd server program's protocol parsing logic. White nodes correspond to nodes which exist on some path between the source and destination nodes. Black nodes correspond to "reject" nodes (nodes from which it is no longer possible to reach the destination node). The labeled "source" node roughly corresponds to the start of the tftp server program's protocol parse code and the labeled "target" node, to the code block indicating acceptance of the tftp packet structure. Note that 2 vulnerable strcpy() functions exist within this subgraph. Both were hit during our GA driven search.

write request packet, we were able to test the ability of our GA to learn a valid tftp packet structure. The tftp packet header format is relatively simple. The minimum header length is 4 bytes and it consists of the following fields:

- **Opcode –** The protocol supports 5 opcodes (1=Read request, 2=Write request, 3=Data, 4=Acknowledgement, 5=Error). This is a 2 byte field.
- **Filename –** A variable length ascii sequence.
- **Null Byte –** Null byte following the filename.
- **Mode –** Contains one of the three strings: "netascii", "octet", or "mail" in any combination of upper and lower case.
- **Null Byte –** Null byte following the mode string.

As you can see in Figure 9, we track the evolution of our GA during the successful generation of a valid tftp packet. The strings included in this figure represent the best genomes found by our GA at specific points during the evolutionary process. Note also that these strings are in hexadecimal. Therefore a single byte consists of 2 hexadecimal digits. Because we have also manually reverse engineered the tftp protocol parsing logic, we know that the first check is performed on the length and that the packet parsing logic proceeds from left to right. We outline the evolutionary steps below:

Generation 0: Consists of an initial population of random strings generated by the GA. The "best genome" evolved at the 0th generation (i.e. "2326"), failed to evolve a string capable of satisfying tftp's first check for a minimum length string.
Generation 1: Here, the grammar generated a longer string that was accepted and enabled the program to progress deeper into the parsing logic. This longer string was rewarded by the GA with a better fitness and was allowed to reproduce more.
Generation 51: We note that the GA has evolved its first valid opcode (remember valid opcodes range from 1 to 5). The "0005" corresponds to an "error". The remaining characters in the genome are interpreted as a filename string, however, the required null terminator is still missing.
Generation 72: We can see that the GA has evolved a different, valid opcode (a write request). It is still, however, missing the null terminator.
Generation 78: The GA learns that a null terminator must follow the opcode and filename bytes. At this point we have generated a packet with both a valid opcode and an acceptable null terminated filename.
Generation 111: We evolve yet another valid opcode (read request == "0002") and learn that it is valid for the filename to be a null string.
Generation 393: We evolve a packet with an invalid mode string (the string is correctly null terminated, however, it represents an invalid mode).
Generation 547: The genetic algorithm finds a valid packet structure with a opcode, filename, and mode strings, all in the right positions relative to each other.

The evolutionary process outlined above is consistent with the process observed in all of our trial runs. Our GA begins with no information about the packet structure, but over successive generations, it incrementally learns what an accepted tftp packet input looks like. Typical running times were on the order of 20-30 minutes. If we allow our GA to continue running, it will quickly generate many unique packets conforming to a valid structure accepted by the tftp program. By collecting enough of them, it may be possible for us to generate an approximate context-free grammar describing the tftp packet specification.

5. Discussion & Limitations

Our technique is limited by the quantity of information embedded in the test program's control flow graph structure. This is due to the fact that we treat each node on the graph as a black box and judge fitness solely based upon runtime execution path information. In a sense, we are performing an "intelligent ", distributed brute force search for the constraints guarding the execution of each node in the control flow graph. Rather than having to satisfy all of the constraints on a given path simultaneously as in a random fuzz, we are able to tackle them one at a time. Thus, it is most useful for code containing a rich, deeply nested control flow structure (e.g. like parser code) and will degenerate to a random bruteforce on flat control flow graphs.

Although an improvement over random input data generation, our technique still suffers from some of the weaknesses inherent to all black box tools. We are able to selectively test interesting regions of program logic and improve our rate of exploration over traditional black box tests, but we cannot guarantee that either a certain rate of coverage will occur or specific destination node will be reached. While we have the capability to reduce the input search space, it can still remain quite large and perform poorly for constraints involving equality tests. Equality tests are more suited "white box" fuzzer's constraint solver.

A closely related limitation concerns our extraction of the control flow graph information. Because we rely upon a valid, static disassembly to obtain the control flow graph, we cannot apply our technique to programs that have been compressed, encrypted, or otherwise obfuscated. We also may miss control flow information that is determined at runtime (e.g. runtime calculations of an index into a call table). Though our initial results are promising, we need to perform additional tests to see how well our methodology will scale to larger programs and more complex protocols. Finally, the approach requires a human analyst to identify an initial source / destination pair describing the region of code to be tested. In the future, this selection might be able to be partially automated (by suggesting regions around known vulnerable API functions, for example).

Some may argue that new "white box" fuzz testing tools will quickly render black box approaches obsolete. We do not feel that is the case. While automated "white box" testing tools theoretically have the ability to test all program paths, they suffer from practical limitations. As a result, they are likely to retain a place in the software vulnerability testing process for quite some time to come.

6. Related Work

A number of researchers have done work in the area of fuzz testing. In the early 1990's Barton Miller et al. [11] first presented the "fuzzing" concept by performing tests on UNIX applications with random inputs. [4] presented fuzz testing on Windows NT GUI based applications. Building upon Miller's work, later researchers successfully applied fuzzing to other forms of input, like network protocols and popular file formats. Random input injection has resulted in the discovery of subtle parsing errors leading to dangerous vulnerabilities. Unfortunately, black box fuzzers have difficulty achieving good code coverage and penetration depth into a program's control flow logic. Later researchers reasoned that incorporating knowledge of the protocol into the input selection process might be more effective than supplying entirely random input. Thus, the idea of using a partially specified or semi-random input structure emerged [8].

Recently, "white box" fuzzers have emerged onto the automated vulnerability analysis scene. Some of the early work in this area was performed by Cader et al. and published in the paper "EXE: Automatically Generating Inputs of Death" [CGP+06]. The white box fuzz testing approach involves symbolically running an application and solving constraints its control graph. The generated constraints are then used to produce new inputs that enable the program to explore new paths. The DART (Directed Automated Random Testing) and SAGE (Scalable, Automated, and Guided Execution) projects are also based upon this idea [6] [5]. In theory, such "white box fuzz testing" approaches seem hard to beat. They are, however, constrained by some practical limitations [5]. These include path explosion, imperfect symbolic executions, and performance bottlenecks relating to the computational expense of constraint solving [5]. They are also substantially much more complex to develop.

There has also been research on the applications of evolutionary computation to the software testing domain. Cheon and Kim proposed a specification based fitness function for testing object oriented programs [1]. Khor and Grogono proposed using data dependency analysis to automatically generate branch coverage test data. [10]. Finally, Minn and Holcombe discussed the applications for the concept of "chaining" in the design of a genetic algorithm based test data generator [12]. Among these existing fitness functions, we believe our application of a Dynamic Markov Model heuristic to the problem of guided input selection is unique and potentially beneficial to the development of future automated vulnerability analysis tools.

7. Conclusions & Future Work

In this paper, we have discussed a new black box fuzzing methodology based upon a dynamic Markov Model heuristic. Our experiments validated our approach. We also demonstrated that this approach can be implemented as a plug in for a commonly used reverse engineering framework, run on an inexpensive platform in a very modest amount of time, and produce practical results on a commercial server application. It consistently outperformed a random fuzzer, especially for greater control flow penetration depths. Our research incorporates ideas from machine learning, statistical theory, static and dynamic software analysis, and reverse engineering. Because of this, it benefits from the synergy of a truly interdisciplinary approach and bridges a gap between theoretical and industrial security research.

There is still much to be done to practically and cost-effectively deploy our system. Also, it is possible to extend our work to other problem domains. Here, we list some possible extensions to this research:

- Testing using other applications and protocols.
- Automating grammar generation by deriving grammar production rules from the strings contained in a target binary.
- Extending our approach to handle protocols with state information (for example, a handshake)
- Because we can generate multiple, unique inputs capable of crashing the program, we could extend our tool for intrusion detection by creating signatures based on those inputs.

7. References

[1] Y. Cheon and M. Kim. *A specication-based fitness function for evolutionary testing of object oriented programs*. Proceedings of the 8th annual conference on Genetic and evolutionary computation, pp. 1953-1954, 2006.

[2] R.C. Collins and J.J. O'Neill. *Grammatical Evolution: Evolving Programs for an Arbitrary Language*. Lecture Notes in Computer Science 1391. First European Workshop on Genetic Programming, 1998.

[3] A. Ethem. *Introduction to Machine Learning*. Boston, Mass.:MIT Press, 2004.

[4] J.E. Forrester and B.P. Miller, *An Empirical Study of the Robustness of Windows NT Applications Using Random Testing*. 4th USENIX Windows Systems Symposium, Seattle, August 2000.

[5] P. Godefroid, M. Levin, D. Molnar. 2007. *Automated Whitebox Fuzz Testing.* Technical Report: Microsoft Research, May 2007.

[6] P. Godefroid, N. Klarlund, and K. Sen. *DART: Directed Automated Random Testing*. In Proceedingsof PLDI'2005 (ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation), pages 213–223, Chicago, June 2005.

[7] Hoglund, G., and McGraw, G. 2004. *Exploiting Software: How to Break Code.* Boston, Mass.:Addison-Wesley.

[8] A. Helin, J. Viide, M. Laakso, J. Röning. *Model Inference Guided Random Testing of Programs with Complex Input Domains.* 2006.

[9] IDA Pro. Web page: http://www.datarescue.com/

[10] S. Khor and P. Grogono. *Using a genetic algorithm and formal concept analysis to generate branch coverage test data automatically*. Automated Software Engineering, 2004. Proceedings. 19th International Conference on, pages 346-349, 2004.

[11] B.P. Miller, L. Fredriksen, and B. So. *An Empirical Study of the Reliability of UNIX Utilities*. Communications of the ACM 33, 12 (December 1990).

[12] P. McMinn and M. Holcombe. *Evolutionary testing of state-based programs*. Proceedings of the 2005 conference on Genetic and evolutionary computation, pp. 1013-1020, 2005.

[13] Nist. Web page: http://nvd.nist.gov/

[14] Pedram A. *PaiMei - Reverse Engineering Framework*, RECON Conference 2006.

[15] Markov chain. Web page: http://en.wikipedia.org/wiki/Markov_chain

[16] C. Ryan, J.J. Collins, M. O'Neill. *Grammatical Evolution. Evolving Programs for an Arbitrary Language*. Lecture Notes in Computer Science 1391. First European Workshop on Genetic Programming 1998.

[17] Wikipedia. *Trivial File Transfer Protocol.* Web page: http://en.wikipedia.org/wiki/Tftp