Protecting Privacy in Key-Value Search Systems

Yinglian Xie Michael K. Reiter David O'Hallaron
Carnegie Mellon University
Email:{ylxie, reiter, droh}@cs.cmu.edu

Abstract

This paper investigates the general problem of efficiently performing key-value search at untrusted servers without loss of user privacy. Given key-value pairs from multiple owners that are stored across untrusted servers, how can a client efficiently search these pairs such that no server, on its own, can reconstruct the key-value pairs?

We propose a system, called Peekaboo*, that is applicable and practical to any type of key-value search while protecting both data owner privacy and client privacy. The main idea is to separate the key-value pairs across different servers. Supported by access control and user authentication, Peekaboo allows search to be performed by only authorized clients without reducing the level of user privacy.*

1 Introduction

Wide area distributed systems often assume that hosts from different administrative domains will collaborate with each other (e.g., [20, 33]). With user data exposed to heterogeneous, third-party servers, one major challenge is to store and find information without loss of privacy.

Consider a distributed service discovery system with multiple independent service providers [7]. Each provider stores service attributes, prices, and locations at one or more directory servers. Clients submit service attributes as queries to the directory servers, and obtain price and location information as query results. This poses a significant risk to the privacy of both the clients and the service providers. A curious directory server could not only follow a client's queries and infer the client's activities, but also exploit the information stored by a service provider to infer sensitive information such as the provider's marketing strategies and financial status.

As another example, consider a people location service for ubiquitous computing environments (e.g., [13]). Although there are many solutions (e.g., [17, 23]) to prevent unauthorized access to user location information, few of them tackle the problem of protecting user privacy with respect to the servers, which may belong to different organizations and be untrusted to expose either data or queries.

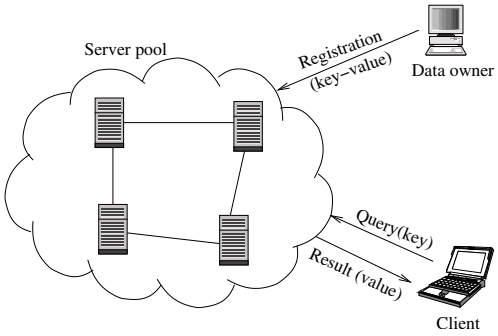

Figure 1. A typical key-value search system

The question then is how can we efficiently search information while protecting the privacy of both data owners and clients? Without loss of generality, in a key-value search system illustrated in Figure 1, there are data owners, clients, and a pool of servers. Data owners register their data represented as key-value pairs at one or more servers. Clients submit keys as queries and would like to retrieve all the values that match the keys. In such a scenario, given key-value pairs from multiple data owners that are stored across untrusted servers, how can a client search keys for values in such a way that no server, in isolation, can infer what the client has queried and retrieved? Meanwhile, we would like no server to be able to determine the key-value bindings stored by any data owner. Figure 2 lists some concrete example key-value pairs in our everyday life.

Prior research on privacy-preserving search has largely focused on providing strong security guarantees. They usually incur high overhead, or provide limited search functionality or privacy that limits their real-world adoptability. For example, PIR approaches (e.g., [5, 14]) can theoretically support key-value search under strong privacy, but with high overhead that has precluded their use in practice. Encryption-based solutions (e.g., [30, 3]) allow clients to search over encrypted data, but limit search to be performed by either clients who hold the same encryption keys

Key	Value	Application
Product names	Providers, prices	Online shopping
Keywords	File owners, file names	Keyword search, file sharing
Stock names	Stock quotes	Stock quote dissemination
Patient names	SSN, medical histories	Online medical directories
Subscriber names	Phone numbers	Yellow-page service

Figure 2. Example applications of key-value search

as the data owners, or on a small number of keywords prespecified by clients. In addition, they often require a sequential scan through the encrypted data and are not efficient. Anonymity-based approaches (e.g., [6, 27]) can also achieve client privacy by routing queries through an anonymous overlay toward servers. These approaches focus on providing anonymity to the clients, but do not protect the privacy for the generalized key-value search.

In this paper, we present the *Peekaboo* system for performing key-value search at untrusted servers without loss of user privacy. We explicitly consider the tradeoffs between privacy, usability, and efficiency. Although there are solutions that achieve strong security properties, we intentionally favor an efficient and practical approach that offers only weaker security properties. Our main idea is to split the key-value pairs across multiple *non-colluding* servers that do not share inputs with each other. All the servers then jointly perform search to return query results. In summary, the Peekaboo system has the following features:

- *Secure:* Given a client query expressed as a key, Peekaboo servers return a list of values matching the key while no server, on its own, can determine either the values retrieved, or the key-value bindings. Therefore, Peekaboo protects both data owner privacy and client privacy. Furthermore, the Peekaboo access control and user authentication mechanisms prevent unauthorized users from searching the data.

- *Flexible:* Peekaboo is applicable to any type of key-value search using user defined match criteria (e.g., exact match [10], range search [12]). It can be easily extended to support advance queries where not only matched values but also matched keys will be returned in query results (e.g., fuzzy match [11]).

- *Efficient:* Peekaboo requires neither expensive routing mechanisms to send data (or queries), nor specialized encryption algorithms on stored data. Our performance evaluation shows that the storage costs of Peekaboo servers are comparable to or even less than legacy servers, whereas the search latency is on the order of tens to hundreds of milliseconds, acceptable to most clients.

2 Model and Definitions

In this section, we describe our system model and the privacy properties that Peekaboo is trying to achieve.

2.1 System Model

The system has three types of entities: data owners (owners hereafter), clients, and Peekaboo servers. We view the data as a list of key-value pairs. Without loss of generality, we assume keys alone do not release useful information about the key-value pairs that are to be searched (i.e., we should not be able to infer a key-value pair from just the key for the purpose of search to be meaningful). Peekaboo servers can store key-value pairs provided by multiple independent owners. A query consists of a single key and the client is interested in retrieving all the values that match the key in the key-value pairs using application specific match criteria (e.g., exact match, range match).

The Peekaboo search protocol consists of two stages: a registration stage and a query stage. In the registration stage, owners publish key-value pairs at Peekaboo servers. In the query stage, clients interact with servers to resolve queries.

The system has two types of Peekaboo servers: *K-servers* and *V-servers*. K-servers store keys only, whereas V-servers store encodings of values that can be used to recover values in the key-value pairs after search. Data owners and clients talk only to the V-servers. Both types of servers jointly perform search to resolve queries. Without loss of generality, we assume: (1) Peekaboo servers are "honest but curious". They follow protocol specifications exactly, and passively observe the information stored locally and the messages they received. (2) Peekaboo servers do not collude to learn data and queries. This does not prevent the servers from communicating with each other in order to follow the protocol. In particular, each K-server authenticates the V-servers, and interacts with only the V-servers that it has authenticated through a protected channel such as TLS [32].

2.2 Privacy Properties

Privacy is a guarantee that certain information about an entity is hidden from other entities. The privacy property is the definition of what types of information are hidden from which entity. In a Peekaboo search system, there are two types of entities whose privacy we would like to protect: data owners and clients.

Throughout both registration and query stages, we strive to prevent K-servers from learning values or user identities. Although K-servers have access to keys, we protect the privacy of data owners and clients by providing anonymity to both of them against the K-servers. Meanwhile, we strive to leak no information about keys or values to V-servers, thus providing confidentiality of both the key-value pairs published by the owners and the key-value pairs retrieved by the clients from the V-servers. Each server, on its own (i.e., without any input from other servers), cannot determine the key-value bindings either stored or queried. Accordingly, we define the following privacy properties for data owners and clients, respectively:

- *Owner privacy:* During both the registration and query stages, a K-server, on its own, should not learn the owner identity and the list of values in the key-value pairs. A V-server, on its own, should not learn either the keys or the values in the key-value pairs.
- *Client privacy:* During the query stage, a K-server, on its own, should not learn the client identity and the list of values returned in the query results. A V-server, on its own, should not learn the client's queried keys or the values retrieved.

Given such privacy definitions, we first describe in Section 3 a basic protocol for performing registration and query with a single K-server and a single V-server. We use this basic protocol as a building block, and present in Section 4 the Peekaboo search system for an open environment, where any client interested in retrieving key-value pairs can perform search. In such a scenario, while we assume servers do not collude with each other, they could actively participate in search as well, performing on-line dictionary attacks by enumerating all possible keys as queries. However, we limit such dictionary attacks to be on-line so that they can be detected and stopped. To prevent such dictionary attacks, we further present in Section 5 an enhanced protocol that limits search to only authorized clients using access control and user authentication mechanisms.

In our model, we achieve a tradeoff between the level of privacy and the usability and efficiency obtained in the protocols. For this purpose, we believe our privacy definitions in an honest-but-curious model is sufficient. We discuss deployment issues and solutions to mitigate server collusion in Section 6. However, if strong privacy is a concern, then more secure protocols such as [19] can be used in the context of keyword-based PIR.

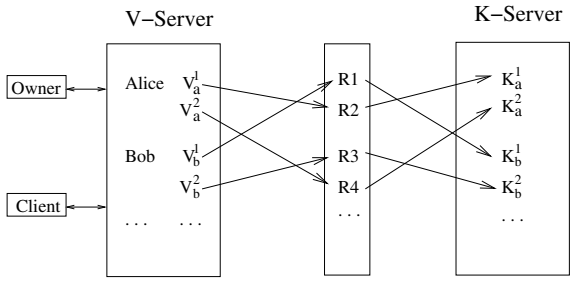

Figure 3. Using rendezvous numbers to bind the keys and the values

3 The Basic Protocol

In this section, we describe a basic protocol as a building block for our system. This basic protocol involves a single K-server and a single V-server, and is based on public key cryptography. For the moment, we assume owners and clients use this basic protocol to publish key-value pairs and to perform search. Since this protocol has only a limited privacy guarantee, we present in the next section how we can construct search systems based on this protocol to provide the desired privacy properties. For clarity, we use upper case $K_1, K_2, \ldots$ to denote keys in the key-value pairs, and use lower case $k_1, k_2, \ldots$ to denote encryption keys that will be needed.

For the specific application of key-value search, keys alone do not release useful information about the pairs for the search to be meaningful. Thus our idea is to split the pairs and store them at different servers by introducing a layer of indirection in between. Figure 3 shows the high level concept of the basic protocol, where owners store the keys at only the K-server and the values at only the V-server. To bind the keys and the corresponding values, we generate a list of *rendezvous numbers* to serve as an indirection layer. Each key-value pair is associated with a unique rendezvous number generated randomly by the V-server, and forwarded to the K-server.

Owners and clients both communicate with only the V-server to publish data and to perform search. Given a client query, both servers work jointly to look up query results using rendezvous numbers. During the communication, keys will be forwarded to the K-server without being exposed to the V-server, whereas the values are stored and returned by the V-server. To simplify our description, we assume the system has a single owner Alice who wants to register a list of key-value pairs $\langle K_1, V_1\rangle, \ldots, \langle K_n, V_n\rangle$, and a single client Charlie who wants to retrieve the value correspond-

ing to a key K_s. The K-server's public key is pk. The basic protocol works as follows (illustrated in Fig. 4):

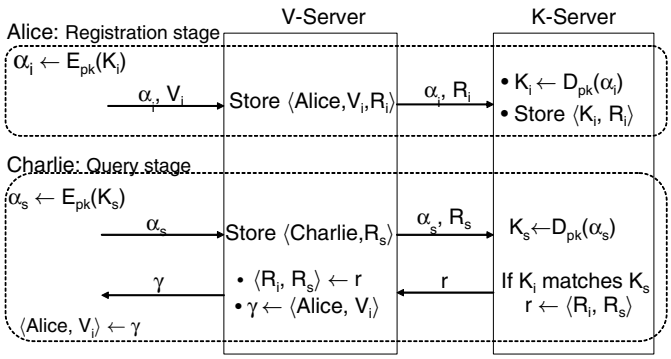

Figure 4. The basic Peekaboo protocol

Registration stage:

Step 1*:* To publish a key-value pair $\langle K_i, V_i\rangle$, Alice encrypts the key K_i with the K-server's public key pk, and submits the encryption $\alpha_i \leftarrow E_{pk}(K_i)$ and the corresponding value V_i to the V-server:

$$\text{Alice} \rightarrow \text{V-server} : \alpha_i, V_i$$

Step 2*:* On reception of the registration request, the V-server extracts V_i and the owner identity "Alice", generates a unique rendezvous number R_i, and stores the following entry locally:

$$\text{V-server} : \langle \text{Alice}, V_i, R_i\rangle$$

The V-server then forwards α_i to the K-server, attaching R_i:

$$\text{V-server} \rightarrow \text{K-server} : \alpha_i, R_i$$

Step 3*:* The K-server extracts $K_i \leftarrow D_{pk}(\alpha_i)$ from the message, where $D_{pk}(\alpha_i)$ denotes the decryption of α_i using the private key corresponding to the K-server's public key pk. It then registers the tuple $\langle K_i, R_i\rangle$ locally:

$$\text{K-server} : \langle K_i, R_i\rangle$$

Query stage:

Step 1*:* To search based on a key K_s, the client Charlie encrypts K_s with the K-server's public key pk, and submits the encryption $\alpha_s \leftarrow E_{pk}(K_s)$ as the query to the V-server:

$$\text{Charlie} \rightarrow \text{V-server} : \alpha_s$$

Step 2*:* The V-server generates a unique rendezvous number R_s for the query, and registers the tuple of the client identity "Charlie" and R_s locally:

$$\text{V-server} : \langle \text{Charlie}, R_s\rangle$$

The V-server then attaches R_s to the original query α_s, and submits a search request to the K-server:

$$\text{V-server} \rightarrow \text{K-server} : \alpha_s, R_s$$

Step 3*:* On reception of the search request, the K-server extracts the queried key $K_s \leftarrow D_{pk}(\alpha_s)$ using its private key. The K-server then performs search locally. If a key K_i matches the query K_s based on the predefined application match criteria (e.g., numerically equal or string match), the K-server returns the tuple $r \leftarrow \langle R_i, R_s\rangle$ as the query result to the V-server, meaning the key with rendezvous number R_i matches the key with rendezvous number R_s:

$$\text{K-server} \rightarrow \text{V-server} : r$$

Step 4*:* The V-server extracts $\langle R_i, R_s\rangle \leftarrow r$, looks up the corresponding value V_i and the owner identity "Alice" using R_i, and returns the final query result $\gamma \leftarrow \langle \text{Alice}, V_i\rangle$ to Charlie:

$$\text{V-server} \rightarrow \text{Charlie} : \gamma$$

We note that in the query stage, no owner participation is needed to perform search. Because the K-server has access to both the keys stored by the owners and the keys submitted as queries, it can search all key-value pairs registered by different owners using application specific matching criteria.

This basic protocol provides the desired privacy protection from the K-server, which has no information about the values, the data ownership, or the client identities. However, it does not provide privacy against the V-server. Although the V-server has no access to the keys that are encrypted under the K-server's public key, it may infer the key-value bindings based on the values returned from client queries.

3.1 Supporting Advanced Queries

In many applications, a client may not only want to get a list of values matching the query, but also be interested in seeing the matched keys as well. For example, in a service discovery system, a client searching for printers will be interested in getting all the attributes regarding the list of printers in order to make the best selection. Another example is keyword search where a client is searching for all file names containing the substring "app".

The basic search protocol can support such advanced queries with a slight modification. In order to obtain matched keys in the query results, the client can attach a one-time encryption key protected by the K-server's public key in the query. For better performance, we can use symmetric keys instead of public keys. As shown in Figure 5, before returning the query results, the K-server encrypts the matched keys using the client-provided encryption key sk, and sends the encryption together with the query results to the V-server.

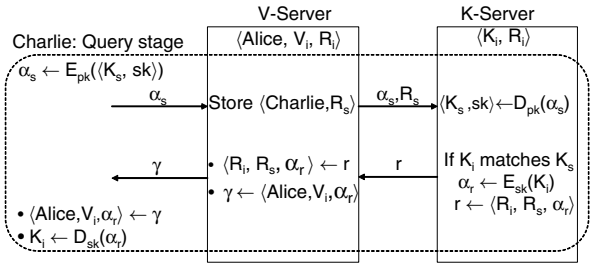

Figure 5. Supporting advanced queries (sk is a one time encryption key generated by Charlie.)

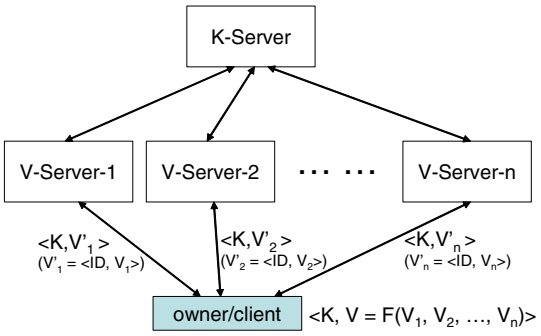

Figure 6. An example Peekaboo search system with a single K-server and n V-servers

4 The Peekaboo Search System

We now consider how we can use the basic protocol to construct a Peekaboo search system with privacy guarantees against both the K-servers and the V-servers. Specifically, we would like to protect the values from being exposed to the V-servers. One option is for each data owner to encrypt the values using a secret key, and store the encryptions at the V-server to return as query results. While clients can still perform search using keys, they need to contact the corresponding owners after search in order to decrypt the returned query results. This increases not only the search latency perceived by clients, but also the processing overhead of data owners, who will be involved in every query that results in a hit from their key-value pairs.

Instead, our idea for preventing the V-servers from accessing the values is to divide every value V in the key-value pairs into n ($n > 1$) pieces, and store them across n V-servers, as shown in Figure 6. Each individual piece $V_i (1 \leq i \leq n)$ reveals no information about V, but the knowledge of t or more pieces can be easily used to reconstruct the original V using a reconstruction function F. One choice of F is an XOR function with $t = n$, which offers not only true information-theoretic privacy, but also fast reconstruction due to its simplicity. More generally, we can adopt (t, n) threshold schemes (e.g., [28]) for choosing F.

To register a key-value pair $\langle K, V \rangle$ in a Peekaboo system with n V-servers, the owner first decomposes V into n different pieces of encodings $V_1, V_2, \ldots, V_n$, so that a client can reconstruct V using t out of the n pieces later. To bind the n different pieces of encodings, the owner assigns a unique identification `ID` to this key-value pair, and composes a set of n new key-value pairs $\{\langle K, V_1' \rangle, \langle K, V_2' \rangle, \ldots, \langle K, V_n' \rangle\}$, where $V_i' = \langle \text{ID}, V_i \rangle$ $(1 \leq i \leq n)$. Finally, the owner communicates with all the n V-servers in parallel using the basic protocol described in Section 3, and registers each tuple $\langle K, V_i' \rangle$ by communicating with the i-th V-server $(1 \leq i \leq n)$.

Similarly, to perform search using key K, the client submits the same query of K to randomly selected t V-servers in parallel using the basic protocol. After retrieving the t different pieces of encodings for V from the selected V-servers, the client reconstructs V using the predefined F.

The minimum number of V-servers to be contacted for retrieving and reconstructing a value V in the system should be determined based on the level of privacy required by the corresponding data owner. To prevent the K-server from becoming the system bottleneck, we can also configure the system with multiple K-servers to balance both the registration and query workload.

The storage overhead at each server, determined by the requirement of the basic protocol, is linear in the number of owners and the number of key-value pairs in total. The communication overhead is linear in the number of query results. Both overheads are comparable to legacy servers. Since a client will talk to multiple V-servers in parallel, the search latency will be slightly higher with public key encryption/decryption and one more round-trip communication between the V-servers and the K-server. We will evaluate the system overhead in Section 7.

5 Access Control and User Authentication

In many applications, data owners would like to control which clients can search which data items. For example, in stock quote dissemination, quotes should be searched by paying customers only. More seriously, without access control and authentication, a malicious user can carefully perform on-line dictionary attacks by searching all possible keys to find out all the registered key-value pairs. We focus on the environments where both users and servers may belong to different organizations and administrative domains. Unlike traditional mechanisms, Peekaboo access control and authentication should be both privacy-preserving and convenient for search to be used frequently. Our design is guided by the following principles:

- *Inter-operability and expressivity.* The system should support users from different organizations or domains.

Given a query, servers should return all query results (which may be from different owners) that the client is authorized to see. Each owner should be able to specify which client can access which key-value pairs based on different levels of data sensitivity.

- *Privacy non-disclosure.* Servers should not be able to infer the key-value pairs from the access control and user authentication information.

- *Convenience to the user.* For convenient practical use, clients should not need to know which data owners can potentially satisfy their queries prior to search. Owners should be able to revoke existing access permissions of their data easily.

With all users talking only to the V-servers, the natural choice is to authenticate users at the V-servers. Access control can be enforced at both the K-servers and the V-servers. For inter-operability, Peekaboo access control and user authentication are based on public key cryptography and we assume an available public key infrastructure (e.g., [22]).

5.1 V-server based Access Control

Given a query, V-servers can perform access control by jointly returning only the values that a client is authorized to see. In the registration stage, each owner creates an Access Control List (ACL) for every $\langle K, V\rangle$ of their key-value pairs, specifying a list of clients that can access the pair. The owner then splits V into n pieces $V_1, V_2, \ldots, V_n$, attaches the ACL with every V_i, and registers the tuple $\langle K, \{V_i, ACL\}\rangle$ by communicating with the i-th V-server using the basic protocol. During the query stage, given all the matched rendezvous numbers returned from the K-server, each V-server returns to a client only the value pieces that the client is allowed to access based on the corresponding ACLs.

However, such V-server based access control may create a side channel of information leakage, where the V-servers may infer the corresponding queried keys based on the number of entries matched and returned from the K-servers. For example, the V-servers can guess whether a searched key is about a popular product with many matching results.

5.2 K-server based Access Control

The K-servers can perform access control as well to prevent the aforemention side channel of information leakage and thus provide a stronger privacy guarantee. By returning only the entries that a client is authorized to access, the K-servers prevent the V-servers from learning the exact numbers of matched results.

However, because the K-servers do not have access to the client identity information, they cannot perform permission checks directly by attaching an owner specified ACL for every stored key. We thus need a solution that can hide client identities in ACLs while still enforcing access control.

Our key idea is to let each owner create a pseudonym C' for a client C whom the owner is granting access. Every such $\langle C, C'\rangle$ client-pseudonym mapping is split in half and stored at different types of servers. In particular, the V-servers authenticate clients with their identity information, while the K-servers check access permission using client pseudonyms to preserve privacy. To prevent a malicious client from forging pseudonyms or using other people's pseudonyms, we use noninteractive zero-knowledge proofs so that the V-servers can ensure the client-submitted pseudonyms are indeed those specified by the data owners, even though the V-servers themselves do not have access to the pseudonyms. We describe the modified protocol with access control between a K-server and a V-server using the same example described in Section 3. The scheme can be easily generalized to a system with multiple V-servers.

Registration stage:

Step 1*:* For each key-value pair $\langle K_i, V_i\rangle$, the owner Alice creates an access control list ACL_i consisting of a list of clients $\{C_1, C_2, \ldots\}$ that can search the pair:

$$\begin{aligned}\text{Alice} \quad : \quad & \langle K_i, V_i, ACL_i\rangle \\ & ACL_i = \{C_1, C_2, \ldots\}\end{aligned}$$

For each client C_i in ACL_i, Alice creates a pseudonym C_i', and replaces C_i with C_i' in ACL_i:

$$\text{Alice} \ : \ ACL_i' = \{C_1', C_2', \ldots\}$$

To register $\langle K_i, V_i\rangle$ with access control information, Alice encrypts both the key K_i and the corresponding ACL_i' using the K-server's public key pk into $\alpha_i \leftarrow E_{pk}(K_i, ACL_i')$, so that only the K-server will be able to see the key and the access control specification.

For each client C_i, Alice constructs a tuple $m_i = \langle C_i, e_i, h_i\rangle$, where $e_i \leftarrow E_{pc_i}(C_i')$ is an encryption of the client pseudonym C_i' using C_i's public key pc_i, and $h_i \leftarrow H(C_i')$ is generated using a one-way hash function H. This tuple m_i allows the V-server to verify the client C_i's input later (we discuss how to choose the hash function later).

Finally, Alice submits α_i, the corresponding value V_i, and the set of tuples $M = \{m_1, m_2, \ldots, \}$ to the V-server:

$$\text{Alice} \rightarrow \text{V-server} \quad : \quad \alpha_i, V_i, M$$

Step 2*:* On reception of the registration request, the V-server generates a rendezvous number R_i, and registers the

following tuple locally:

$$\text{V-server} \quad : \quad \langle \text{Alice}, V_i, R_i, M \rangle \\ \text{where } M = \{\langle C_1, e_1, h_1 \rangle, \langle C_2, e_2, h_2 \rangle, \ldots\}$$

The V-server then forwards the encryption α_i as well as the rendezvous number R_i to the K-server:

$$\text{V-server} \rightarrow \text{K-server} : \alpha_i, R_i$$

Step 3*:* The K-server decrypts the message $\langle K_i, ACL'_i \rangle \leftarrow D_{pk}(\alpha_i)$, and registers the data and the access control information locally:

$$\text{K-server} : \langle K_i, ACL'_i, R_i \rangle$$

Query stage:

Step 1*:* To search based on a key K_s, the client Charlie first submits a "ready-to-search" request to the V-server with his identity $I_c =$ "Charlie":

$$\text{Charlie} \rightarrow \text{V-server} : I_c$$

Step 2*:* Given the "ready-to-search" request from "Charlie" $= I_c$, the V-server extracts Charlie's pseudonym from the tuple $\langle \text{Charlie}, e_c, h_c \rangle$ where $e_c = E_{pc}(C')$ and $h_c = H(C')$ if one exists, and presents e_c and h_c to Charlie:

$$\text{V-server} \rightarrow \text{Charlie} : e_c, h_c$$

Step 3*:* Charlie decrypts e_c and finds out his pseudonym $C' \leftarrow D_{pc}(e_c)$.

To send his query K_s and his pseudonym C' to the K-server without leaking the information to the V-server, Charlie re-encrypts both K_s and C' with the K-server's public key pk as $\alpha_s \leftarrow E_{pk}(K_s)$ and $e'_c \leftarrow E_{pk}(C')$, respectively.

In addition, to prove that C' (hidden in the encryption e'_c) is indeed the one returned by the V-server, Charlie constructs a noninteractive zero knowledge proof π that $H(D_{pk}(e'_c)) = h_c$. Charlie then submits α_s, e'_c, and π back to the V-server:

$$\text{Charlie} \rightarrow \text{V-server} : \alpha_s, e'_c, \pi$$

Step 4*:* The V-server first verifies that the proof π presented by Charlie is true based on the received e'_c and the stored h_c. It then creates a rendezvous number R_s, and registers the entry $\langle \text{Charlie}, R_s \rangle$ for this query locally:

$$\text{V-server} : \langle \text{Charlie}, R_s \rangle$$

The V-server then forwards α_s and e'_c to the K-server, attaching the rendezvous number R_s:

$$\text{V-server} \rightarrow \text{K-server} : \alpha_s, e'_c, R_s$$

Step 5*:* On reception of the query, the K-server decrypts α_s and e'_c to obtain both the queried key $K_s \leftarrow D_{pk}(\alpha_s)$ and the corresponding client pseudonym $C' \leftarrow D_{pk}(e'_c)$. The K-server then performs both search and access permission check. Only those query results that are allowed to be accessed by Charlie's pseudonym C', e.g., R_i, will be returned as $r \leftarrow \langle R_s, R_i \rangle$ to the V-server:

$$\text{K-server} \rightarrow \text{V-server} : r$$

Step 6*:* Finally, the V-server looks up the values based on the K-server returned rendezvous numbers, and sends the query results $\gamma \leftarrow \langle \text{Alice}, V_i \rangle$ back to Charlie:

$$\text{V-server} \rightarrow \text{Charlie} : \gamma$$

Discussion

By verifying the noninteractive zero-knowledge proof π that $H(D_{pk}(e_c)) = h_c$, the V-server is convinced that a client is using the pseudonym sent to it by the V-server when the client submits a query. The form of this proof depends on the form of encryption and the one-way function H, but certain such functions permit π to be constructed at a computational expense roughly equal to the expense of a digital signature. In one example, the K-server selects a cyclic group $\mathcal{G}$ in which both the Decisional Diffie-Hellman problem and computing square roots are intractable, and utilizes Shoup-Gennaro encryption [29] in $\mathcal{G}$ and one-way function $H : \mathcal{G} \rightarrow \mathcal{G}$ defined by $H(x) = x^2$; see, e.g., [25, Section 5.2] for details.

Note access control checking is only performed on private data. For public data that can be searched by anonymous clients, Alice simply tags them as "public" at both the K-server and the V-server for better search performance.

To support *groups*, Alice can create a pseudonym G' for each group G in ACL specification. For each member C_i in G, Alice encrypts the group pseudonym G' using C_i's public key pc_i, and obtains the encryption $e_i \leftarrow E_{pc_i}(G')$. Finally, Alice computes the hash $h_i \leftarrow H(G')$, and sends $m_i = \langle C_i, e_i, h_i \rangle$ to the V-server during the data registration, so that C_i can use G' to perform search later:

$$\text{Alice} \rightarrow \text{V-server} : m_i$$

To revoke a client C_i's access rights on a particular key-value pair $\langle K_j, V_j \rangle$, Alice simply removes C_i's pseudonym C'_i from the corresponding ACL'_j at the K-server. Such permission revocation can take effect immediately without being noticed by the client at all.

Since each owner selects client pseudonyms independently, a client may need to decrypt multiple different pseudonyms from different owners during the query stage. To reduce the query overhead, pseudonyms can be cached at the client side in the first query, and reused at subsequent

queries to avoid the first two steps in the query stage. Alternatively, each client can select a unique pseudonym (e.g., a user ID), and register it at different owners for permission specification.

5.3 User Authentication

For inter-operability, the Peekaboo user authentication is based on conventional digital signatures (e.g., [9]). To defend against replay attacks, we use timestamps and assume loosely synchronized clocks. When submitting a query to the V-server in *step* 3, the client Charlie generates a timestamp T, signs T and the rest of query, and submits the following message including the signature to the V-server:

$$\text{Charlie} \rightarrow \text{V-server} \quad : \quad Q, \sigma$$

where $Q = \{T, \alpha_s, e'_c, \pi\}$. Here $\alpha_s \leftarrow E_{pk}(K_s)$ and $e_c \leftarrow E_{pk}(C')$. σ is the digital signature of Q.

On reception of the message, the V-server verifies the signature σ using Charlie's public key, which can be obtained from a public key infrastructure. The V-server then processes the query using the procedures described above.

In summary, the revised Peekaboo protocol with access control and user authentication is illustrated in Figure 7.

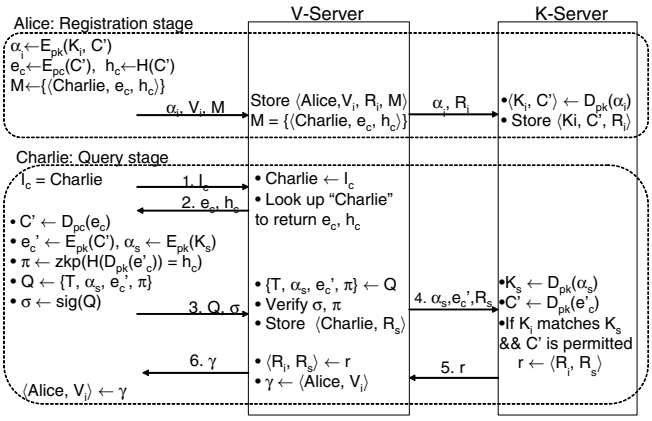

Figure 7. The enhanced protocol with access control and user authentication, where "zkp" is an abbreviation for "zero knowledge proof".

6 Deployment and Vulnerabilities

In this section, we discuss various issues in system deployment, and outline potential malicious attacks that Peekaboo is vulnerable to with possible solutions. Completely addressing these attacks is a topic of ongoing work.

A basic assumption of the Peekaboo search system is that the K-servers have no information about user identities or data values. Therefore, in real deployment, the K-servers should not have access to the network packets routed toward the V-servers, for example, not be configured on transit network backbones.

Both the K-servers and the V-servers could misbehave by producing arbitrary or bogus search results. To detect misbehaving servers, we can use both owner-initiated auditing and client-initiated auditing based on random sampling so that the more the server misbehaves, the higher the probability that it will be caught. For server non-repudiation, both servers can sign their responses in query results.

A more serious threat is server collusion, where the K-servers and the V-servers cooperate to reconstruct the key-value pairs registered or searched. One approach is to introduce further layers of indirection by adding auxiliary servers in the system. Specifically, we can deploy a chain of auxiliary servers between a K-server and a V-server to perform the basic Peekaboo protocol. A registration request $\langle \alpha, V\rangle$ ($\alpha \leftarrow E_{pk}(K)$) is first submitted to a V-server, who creates a rendezvous number R_0 and forwards $\langle \alpha, R_0\rangle$ to the first auxiliary server in the chain. Each auxiliary server A_i, on reception of the message, randomly generates a new rendezvous number R_i to replace the old one R_{i-1} in the message to forward to the next hop server, until the request finally reaches the K-server. Similarly, a user query is also routed incrementally along the server chain from the V-server to the K-server. Query results are then propagated back in the reverse direction. To tolerate the brute force collusion of up to t servers in this chain, we need at least $t - 1$ auxiliary servers between the K-server and the V-server. There is thus a balance between privacy and efficiency, and both the data owners and clients can jointly decide the level of desired privacy.

However, such multi-server protocol is vulnerable to timing attacks with the collusion between the K-server and the V-server. For example, both servers could jointly measure the time needed between the V-server forwarding a query and the K-server receiving it, or submit queries one by one to learn who submitted what query. To mitigate such attacks, we can use solutions from anonymous routing [4, 31]. For example, each auxiliary server can buffer and reorder messages within a small time frame.

Finally, Peekaboo servers could perform traffic pattern analysis to infer popular keys or values by measuring the frequency of the corresponding rendezvous numbers in queries or query results. In particular, the V-servers could tell whether two queries resulted in the same response even though they have access to neither queries nor query results. To mitigate this threat, owners can initiate the registration process frequently in order to update both the rendezvous numbers and the encodings of values held by the V-servers.

7 Example Applications and Performance

In this section, we describe an example application of a file sharing service to illustrate how Peekaboo can be used to perform keyword search without loss of user privacy. We then evaluate the protocol overhead using trace-based experiments and compare its performance with regular centralized servers.

In a file sharing system, owners store files or file names at directory servers. Each file has an owner-assigned local ID. Clients submit queries as keywords to the servers. If the content or the name of a file matches the query, the servers return the local file ID and the corresponding owner identity (e.g., IP address) as query results. Clients can then download the file directly using the local file ID from the corresponding file owner.

In the Peekaboo system registration stage, owners register the file names (or file content) as the keys, and the local file IDs as the values at the K-servers and the V-servers, respectively. For each file, the V-servers randomly generate unique 128-bit strings as the rendezvous numbers. To support efficient query search, each V-server computes a hash based inverted index of rendezvous numbers, whereas each K-server computes an inverted index table of keywords. In the query stage, clients submit keywords as the queried keys, and get a list of matched values represented as the local file IDs and the corresponding file owner IP addresses. When advanced queries are supported where matched keys should be returned in query results, the servers also return the list of matched file names (or relevant file content) encrypted by the client-provided one-time encryption key.

We use a Gnutella [15] trace gathered at CMU to conduct trace based experiments, and evaluate the system performance using the described file sharing application in the following three aspects: (1) the storage costs at both types of servers, (2) the search latency perceived by clients, and (3) the overhead of access control and user authentication. We implemented both types of servers, and evaluate the performance using a single K-server and a single V-server that communicate via the basic protocol, as described in Section 3. Increasing the number of V-servers in the system would not change either (1) or (3). Because clients communicate with V-servers in parallel, the search latency should increase only slightly with multiple V-servers. For comparison, we also implemented a regular centralized server that performs both data registration and query, and repeat our experiments. All the servers are implemented in C++ in Linux, running on PIII 550MHz machines with 128 RAM on a 10BaseT Ethernet LAN. Each data point in the figures below is the average of ten runs.

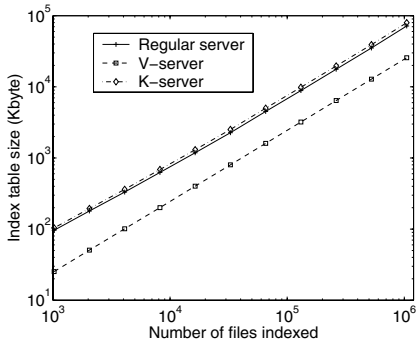

Figure 8. Index table size vs. number of indexed files

7.1 Storage Costs at Peekaboo Servers

To evaluate the storage costs, we extract file names and their owner IP addresses from the search reply messages in the Gnutella trace, and register them using a fake owner program simulating different file owners. Figure 8 shows the index table sizes of the Peekaboo servers and the regular centralized server as the function of the number of indexed files. We observe that the storage costs increase linearly as the number of indexed files increases. The K-server index table sizes are slightly larger compared with a regular server, while the V-server index sizes are only about a third of those of a regular server. In general, the storage costs are small at both types of Peekaboo servers.

7.2 Search Latency Perceived by Clients

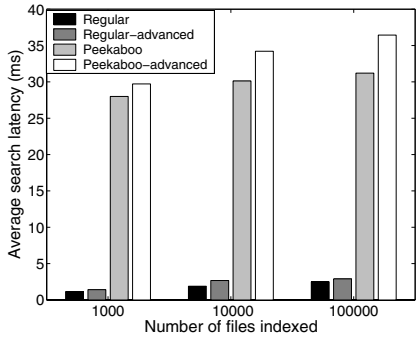

Figure 9. Peekaboo search latency

We proceed to evaluate the search latencies perceived by clients. We implemented a client program running at a third machine (PIII 550MHz with 128 RAM) in the same Ethernet LAN. The public key encryption uses the RSA algorithm [26] with 1024-bit keys, and the one-time symmetric key encryption uses the AES algorithm [1] with 128-bit keys. Both algorithms are implemented by the Crypto++

	Total	Network	Look up	RSA en.	RSA de.	AES en.	AES de.	Other
Mean (μs)	36475	6427	3041	1575	23834	581	781	236
Std dev (μs)	2869	2831	53	10	38	14	1	5
Percentage	100.0%	17.6%	8.3%	4.3%	63.34%	1.6%	2.1%	0.6%

Figure 10. Time to process a search request using 1024-bit RSA keys and 128-bit AES keys.

library (version 4.2) [8]. For each query, the servers return the first 100 matched files as query results. Figure 9 shows the search latencies measured by the client. Compared with the regular server, Peekaboo incurs much higher search latency. When we use the advanced queries to support returning matched keys (i.e., matched file names), the search latency increases only slightly compared with the basic protocol.

To further examine the search latency, we list the times spent in various steps of processing a query in Figure 10. We fix the number of files indexed to be 10^5, and show both the mean and the standard deviation of latency as well as the percentage of the total latency. The "Total" column corresponds to the time elapsed between the client submission of a query and getting back the reply. RSA decryption and network transmission are the most expensive steps, whereas AES encryption and decryption are fast, accounting for less than 5% of the processing time in total. The "Look up" time includes both the K-server lookup and the V-server lookup, and depends on the number of files indexed. The "Other" line consists of the time spent for the V-server to buffer and forward client requests to the K-server as well as the time spent to buffer and forward AES-encrypted replies back to the client. In general, the search latency is acceptable to clients since the network latencies on WAN are usually on the order of tens of milliseconds [21]. By optimizing the security operations (e.g., by using cryptographic routines implemented in hardware), we expect the performance penalties due to security to decrease. Furthermore, if clients will submit multiple queries in a row, they can set up symmetric session keys with the K-server for encrypting/decrypting queried keys to amortize the costs of RSA decryption.

7.3 Overhead of Access control and Authentication

The Peekaboo access control and user authentication mechanisms introduce additional query processing overhead. The V-server based access control is relatively simple and should incur only a small amount of overhead by performing an additional ACL lookup before returning results. We thus consider the K-server based access control described in Section 5.2 to estimate the worst case performance. Such access control and authentication introduce the following extra steps during the query processing: (1) client signature signing and verification, (2) client pseudonym encryption and decryption, and (3) noninteractive zero-knowledge proof construction and verification. While the digital signature based client authentication has a relatively constant cost, the cost of decrypting pseudonyms can grow linearly with the number of client pseudonyms assigned by different owners. Fortunately, such expensive computations are performed by the clients which will less likely become overloaded compared with the servers. In addition, the client pseudonyms can be cached at the client side to reduce the search latency. We implemented both (1) and (2) in the example prototype for our evaluation. The noninteractive zero-knowledge proof, as discussed in Section 5.2, can be constructed with computational expense roughly equal to the expense of a digital signature.

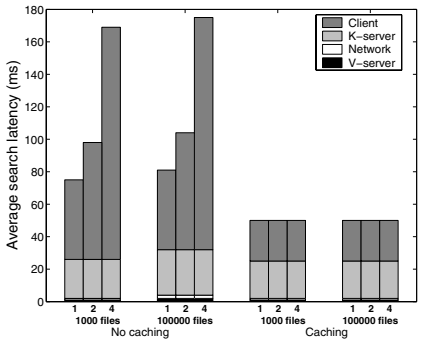

Figure 11. Search latency with access control and user authentication. The client is associated with 1, 2, 4 pseudonyms, respectively

Figure 11 shows the search latency with the Peekaboo access control and user authentication mechanisms by varying the number of indexed files and the number of client pseudonyms. For comparison, we list the processing time spent at various entities as well as the time spent on network transmission. Without pseudonym caching, the client side processing takes the longest time. In general, the increase of the number of files has little effect on search latency. The client side processing time increases proportionally to the number of client pseudonyms, while the server side processing latency increases only slightly with the increasing number of client pseudonyms. By caching client pseudonyms, we can greatly reduce the client processing time, and therefore reduce the overall search latency.

8 Related Work

A number of recent solutions have been proposed to perform search on encrypted data (e.g., [30, 3]). Although encryption provides data confidentiality to protect privacy, it limits the search functionality that can be performed. These approaches either require clients to share the same encryption keys used by the data owners [30], or limit search to be performed on a small number of keywords pre-specified by the clients. They also require a sequential scan through encrypted data for each query, which is an expensive operation in terms of performance.

There has been a large body of literature on anonymous communication to prevent discovery of source-destination patterns. In general, there are two types of approaches: proxy based approaches and mix based approaches. Proxy based approaches interpose one or more proxies between the sender and the receiver to hide the sender's identity from the receiver. Examples include Janus [18] and Crowds [24]. The Peekaboo V-servers bear some similarity with proxies in that all user traffic goes through them. However, the primary purpose of Peekaboo is not to hide user identities, but rather to perform search without revealing the key-value pairs. Thus the Peekaboo V-servers are not just proxies as they actively participate in storing and returning values. The mix based approaches interpose (e.g., [4, 31]) a chain of proxies between the sender and the receiver to achieve unlinkability between the sender and the receiver. We showed in Section 6 where we used mix based approaches to prevent timing attacks. Compared with these approaches, Peekaboo protects not only user identities, but also key-value pairs. However, Peekaboo does not provide unlinkability between key-value pairs in the presence of server collusion.

The problem of Private Information Retrieval (PIR) [5, 14] has been well studied to protect client privacy in search. These approaches model the database as an n-bit string, and a client retrieves the i-th bit without revealing the index i. Although PIR schemes can achieve very strong security, they are generally not practical to use.

Secure multi-party computation (SMC) has also been widely studied [16]. These techniques enable multiple parties, each holding a private input, to collectively compute a function of their inputs while revealing only the function output. Our problem can be viewed as a special case of this problem, though it permits more efficient solutions than general SMC techniques, which are rarely efficient for practical use.

Recent work [2] has also noticed the value of two or more logically independent servers for maintaining the privacy of database content and queries. It envisions an architecture where data and queries can be decomposed across multiple servers in different ways. The authors leave open a concrete solution based on the proposed architecture, and Peekaboo may potentially be adapted to provide a starting point.

9 Conclusion

We have proposed a system called *Peekaboo*, for performing general key-value search at untrusted servers without loss of efficiency and user privacy. Given a set of key-value pairs from multiple owners that are stored across untrusted servers, Peekaboo allows a client to search these pairs in such a way that each server, in isolation, cannot determine any of the key-value bindings. Our main idea is to separate the key-value pairs and store them across different servers. Supported by access control and user authentication, Peekaboo is: (1) secure in that search can be performed by only authorized clients while protecting the privacy of both data owners and clients, (2) flexible in that it is applicable to any type of key-value search, and can be easily extended to support advanced queries, and (3) efficient in that it has small storage cost and search latency, and hence is practical to use today.

References

[1] AES. `http://csrc.nist.gov/CryptoToolkit/aes/rijndael`.

[2] G. Aggarwal, M. Bawa, P. Ganesan, H. G. Molina, K. Kenthapadi, R. Motwani, U. Srivastava, D. Thomas, and Y. Xu. Two can keep a secret: A distributed architecture for secure database services. In *CIDR*, 2005.

[3] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Proc. of Eurocrypt*, 2004.

[4] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–88, 1981.

[5] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *IEEE Symposium on Foundations of Computer Science*, 1995.

[6] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies:International Workshop on Design Issues in Anonymity and Unobservability, LNCS 2000*.

[7] S. Czerwinski, B. Zhao, T. Hodes, A. Joseph, and R. Katz. An architecture for a secure service discovery service. In *Mobile Computing and Networking*, 1999.

[8] W. Dai. Crypto++. `http://www.eskimo.com/~weidai/cryptlib.html`.

[9] Digital signature standard (DSS). *Federal Information Processing Standards Publication 186*, 1994.

[10] Exact match. `http://www.searchenginedictionary.com/e.shtml#exactmatch`.

[11] Fuzzy match. http://www.searchenginedictionary.com/terms-fuzzy-matching.shtml.

[12] J. Gao and P. Steenkiste. An adaptive protocol for efficient support of range queries in DHT-based systems. In *ICNP*, 2004.

[13] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste. Project aura: Towards distraction-free pervasive computing. In *IEEE Pervasive Computing 1*, 2002.

[14] Y. Gertner, Y. Ishai, E. Kushilevitz, and T. Malkin. Protecting data privacy in private information retrieval schemes. *Journal of Computer and System Sciences (JCSS)*, 60(3):592–629, 2000.

[15] Gnutella hosts. http://www.gnutellahosts.com.

[16] O. Goldreich. *The Foundations of Crytography - Volume 2*. Cambridge University Press, 2004.

[17] U. Hengartner and P. Steenkiste. Protecting access to people location information. In *Proc. of the First International Conference on Security in Pervasive Computing*, 2003.

[18] The Lucent personalized web assistant. http://www.bell-labs.com/project/lpwa/history.html.

[19] L. Kissner, A. Oprea, M. K. Reiter, D. Song, and K. Yang. Private keyword-based push and pull with applications to anonymous communication. In *Applied Cryptography and Network Security (ACNS)*, 2004.

[20] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *ASPLOS 2000*.

[21] H. A. Lagar-Cavilla, N. Tolia, R. Balan, E. de Lara, M. Satyanarananan, and D. O'Hallaron. Dimorphic computing. Technical report, Carnegie Mellon University, CMU-CS-06-123.

[22] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Trans. Computer Systems*, 10(4):265–310, 1992.

[23] U. Leonhardt and J. Magee. Security considerations for a distributed location service. *Journal of Network and Systems Management*, 6:51–70, 1998.

[24] M. Reiter and A. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, 1998.

[25] M. K. Reiter, X. Wang, and M. Wright. Building reliable mix networks with fair exchange. In *Applied Cryptography and Network Security (ACNS)*, 2005.

[26] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 27(2), 1978.

[27] V. Scarlata, B. Levine, and C. Shields. Responder anonymity and anonymous peer-to-peer file sharing. In *ICNP 2001*.

[28] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11), 1979.

[29] V. Shoup and R. Gennaro. Securing threshold cryptosystems against chosen ciphertext attack. *Journal of Cryptology*, 15, 2002.

[30] D. X. Song, D. Wagner, and A. Perrig. Practical solutions for search on encrypted data. In *IEEE Symposium on Security and Privacy*, 2000.

[31] P. Syverson, D. Goldschlag, and M. Reed. Anonymous connections and onion routing. In *Proc. of the IEEE Symposium on Security and Privacy*, 1997.

[32] The TLS protocol. http://www.ietf.org/rfc/rfc2246.txt.

[33] M. van Steen, F. Hauck, P. Homburg, and A. Tanenbaum. Locating objects in wide area systems. In *IEEE Communications Magazine*, pages 104–109, 1998.