Protecting Privacy in Key-Value Search Systems

Yinglian Xie David O'Hallaron
Michael K. Reiter

July 2003
CMU-CS-03-158

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

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

We propose a protocol, called *Peekaboo*, that is applicable to any type of key-value search while protecting both the data owner privacy and the client privacy. The main idea is to separate the key-value pairs and store them on different servers based on an important observation that key-value pairs release information only if they are together. Supported by access control and user authentication, Peekaboo allows search to be performed only by authorized clients without reducing the levels of user privacy.

Keywords: key-value search, privacy, untrusted servers, access control, authentication

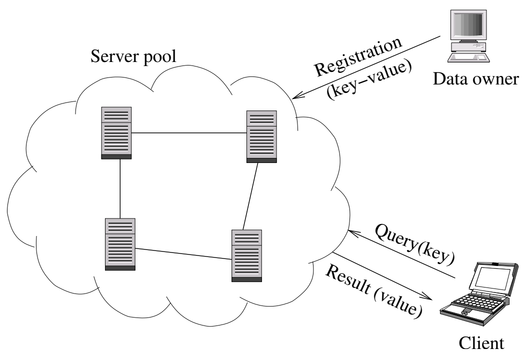


Figure 1: A typical key-value search system

1 Introduction

Wide area distributed systems often assume that hosts from different administrative domains will collaborate with each other [16, 28]. With user data exposed to heterogeneous, untrusted 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 [6]. Each provider stores service attributes or descriptions at one or more directory servers. Clients submit queries to the directory servers for service location information. 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 its access to the service types and prices of the providers to infer more sensitive information such as the company financial status.

A similar problem exists in people location services for ubiquitous computing environments such as Aura [10]. Although there are many results [14, 18, 24] about preventing 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. On one hand, users may not want to expose their location information to the servers. On the other hand, clients would like to keep their queries secret from the servers as well.

As another example, consider a distributed stock quote dissemination system where stock agents publish real time quotes to a number of servers. Clients submit stock names to the servers for the most up to date quotes. Since real time quotes are often billed, stock agents do not want the servers to learn the quotes, whereas clients are not willing to let the servers know which stocks they are interested in.

The question then is how can we efficiently search information while protecting the privacy of the data owners and clients? Without loss of generality, in a key-value search system illustrated in Fig. 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

Key	Value	Application
Keywords	File owners, file names	Keyword search, file sharing
Service attributes	Service locations, providers	Service discovery
User names	Location information	People location service
Stock names	Stock quotes	Stock quote dissemination
Game names	Players, servers	On-line game player match
Colleges, classes	User names	Classmate search

Figure 2: Example applications of key-value search

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 determine any of the key-value bindings? To make the problem more concrete, Fig. 2 lists some example key-value pairs in our everyday life.

In this paper, we propose a protocol called *Peekaboo*, for performing key-value search at untrusted servers without loss of user privacy. An important observation is that key-value pairs release information only if they are together. Thus the main idea of Peekaboo is to split the pairs and introduce a layer of indirection in between. The two different parts of the pairs are stored separately at two non-colluding servers which jointly perform search to return query results. In summary, the Peekaboo protocol 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 the key-value bindings. Therefore, Peekaboo protects both the data owner privacy and the client privacy. Furthermore, the Peekaboo access control and user authentication mechanisms prevent unauthorized users from searching the data without reducing the levels of user privacy.

- *Flexible.* Peekaboo is applicable to any type of key-value search. Given a user query, Peekaboo servers can return the matched values using any user defined searching criteria. It can also 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).

- *Efficient.* Peekaboo does not require expensive routing mechanisms to send data (or queries). Nor does it need specialized encryption algorithms on stored data. The performance evaluation shows that the storage costs of Peekaboo servers are comparable or even less than legacy centralized servers, whereas the search latency is on the order of tens to hundreds of milliseconds, which is acceptable to most clients.

The rest of the paper is organized as follows: Section 2 describes our system model and the privacy properties. In Section 3, we present the Peekaboo search protocol of protecting user privacy. In Section 4, we present the Peekaboo access control and authentication mechanisms. We then discuss the various issues in system deployment including possible malicious attacks in Section 5. Section 6 presents example applications of Peekaboo and the system performance. Section 7 discusses related work before we conclude with Section 8.

2 Model, Definitions and Discussion

In this section, we describe our system model and the privacy properties that Peekaboo is trying to achieve. We also discuss the motivation underlying the definitions and the limitations.

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. We assume either the keys or the values alone do not release useful information about the data (i.e., we should not be able to infer a key-value pair from the key or the value). Peekaboo servers can store data from multiple independent owners. A query consists of a single key and the client is interested in retrieving a list of values matching the key.

The Peekaboo search protocol consists of two stages: registration stage and query stage. In the registration stage, owners publish data at Peekaboo servers. In the query stage, clients interact with Peekaboo servers to resolve queries. Throughout both stages, each server, in isolation, should not be able to reconstruct the key-value pairs published by the owners. Similarly, no server, in isolation, should be able to reconstruct the key-value pairs that are retrieved by the clients.

There are two types of Peekaboo servers: the *K-server* and the *V-server*. The K-server stores keys only, whereas the V-server stores values only. Data owners and clients talk only to the V-server. Both servers jointly perform search to resolve queries. Without loss of generality, we assume: (1) Peekaboo servers are ”honest but curious”. They follow the protocol specifications exactly, and passively observe the information stored locally and the messages they received. (2) Peekaboo server 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.

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 is 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. Fig. 3 shows the types of information that will be learned by each server during the registration and query stages.

Stage	K-server	V-server
Registration	Keys	Values, ownership
Query	Keys	Values, client identities

Figure 3: Peekaboo privacy properties

Throughout both registration and query stages, we strive to prevent the K-server from learning the values and the user identities. And we strive to leak no information about keys to the V-server. Thus each server, on its own (i.e., without any input from the other server), cannot determine the key-value bindings. Accordingly, we define the following privacy properties for data owners and clients, respectively:

- *Owner privacy:* The K-server, on its own, should not learn the owner identity and the list of values in key-value pairs during the registration and query stages. Similarly, the V-server, on its own, should not learn the keys in key-value pairs during the registration and query stages.
- *Client privacy:* The K-server, on its own, should not learn the client identity and the list of values in the query results during the query stage. Similarly, the V-server, on its own, should not learn the client's queried keys during the query stage.

2.3 Limitations

The Peekaboo privacy properties are general, but have limitations.

We note that the data ownership is stored together with the values in the key-value bindings. In general, the data ownership can be regarded as an attribute associated with the values for clients to further communicate with owners after search (e.g., downloading data in keyword search, or accessing services in a server discovery system). Peekaboo requires that values do not leak user privacy when associated with owner identities. For a small number of applications where values should not be associated with the owner identities (e.g., user location service), we can add one more server (e.g., wireless access point) to act as a proxy talking to the V-server on behalf of the users.

The Peekaboo search protocol is vulnerable to on-line dictionary attacks. It allows any client interested in retrieving the key-value pairs to perform search. In some applications, the key space may be small (e.g., stock quote dissemination). Thus a client can enumerate keys in the queries to find out all the stored key-value pairs. We limit such dictionary attacks to be on-line so that they can be detected and stop. To completely prevent dictionary attacks, we should limit the search to only authorized clients using access control and user authentication mechanisms, which will be discussed in Section 4.

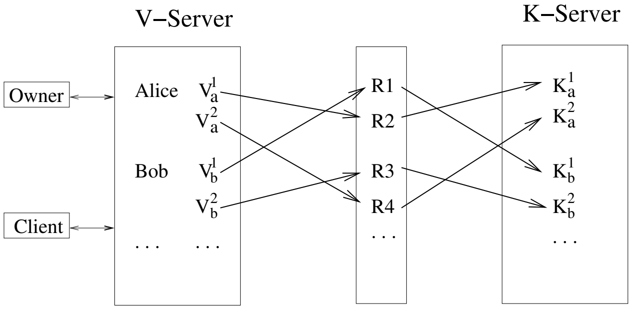


Figure 4: Using rendezvous numbers to bind the keys and the values

3 The Peekaboo Search Protocol

In this section, we describe the Peekaboo search protocol. Given a client query of a key, Peekaboo servers should return a list of values matching the key while each server, in isolation, should not be able to determine the key-value bindings.

An important observation is that key-value pairs release information only if they are together. For example, in stock quote dissemination, neither stock names nor quotes alone are useful. It is the combination of them that matters. Thus the main idea is to split the pairs and introduce a layer of indirection in between. Specifically, we 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 the layer of indirection (see Fig. 4). 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 only with 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.

Next, we first present the basic Peekaboo search protocol, which is based on public key cryptography. We then describe a protocol extension to support advanced queries where not only matched values but also matched keys can be returned in query results (e.g., fuzzy match). 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.

3.1 The Basic Peekaboo Protocol

In the basic Peekaboo protocol, the system is configured with a single V-server and a single K-server. Owners and clients interact only with the V-server. During the communication, keys will be forwarded to the K-server without being exposed to the V-server, while 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 corresponding to a key K_s. The K-server's public key is pk. The protocol works as follows (illustrated in Fig. 5):

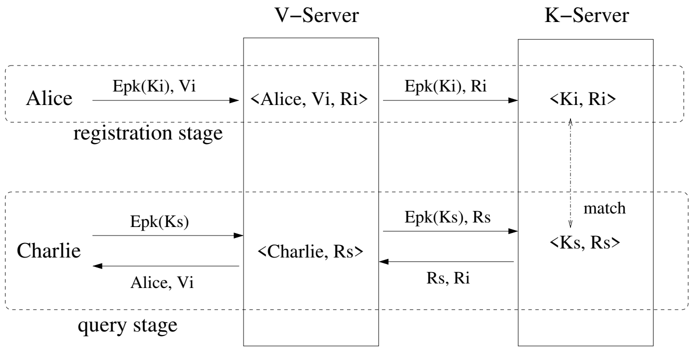


Figure 5: The basic Peekaboo search protocol

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 $E_{pk}(K_i)$ and the corresponding value V_i in plaintext to the V-server:

$$\text{Alice} \rightarrow \text{V-server} \; : \; E_{pk}(K_i), V_i$$

Step 2*:* On reception of the registration request, the V-server extracts the value V_i and the owner identity *Alice* from the message, 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 the encryption $E_{pk}(K_i)$ to the K-server, attaching the newly generated rendezvous number R_i:

$$\text{V-server} \rightarrow \text{K-server} \; : \; E_{pk}(K_i), R_i$$

Step 3*:* The K-server decrypts $E_{pk}(K_i)$ using its private key to get K_i, and 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 $E_{pk}(K_s)$ as the query to the V-server:

$$\text{Charlie} \rightarrow \text{V-server} \;:\; E_{pk}(K_s)$$

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 the rendezvous number R_s to the original query, and submits a search request to the K-server:

$$\text{V-server} \rightarrow \text{K-server} \;:\; E_{pk}(K_s), R_s$$

Step 3*:* On reception of the search request, the K-server decrypts the encryption in the message using its private key, and obtains the queried key K_s. The K-server then performs search locally. If a key K_i matches the query K_s based on some application match criteria (e.g., numerically equal or string match), the K-server extracts the corresponding rendezvous number R_i, and returns the tuple $\langle R_s, R_i \rangle$ as the query result to the V-server, meaning the key with rendezvous number R_i matches the key with rendezvous number R_s:

$$\text{K-server} \rightarrow \text{V-server} \;:\; \langle R_s, R_i \rangle$$

Step 4*:* Given the query result R_i from the K-server, the V-server looks up the corresponding value V_i and the owner identity *Alice* as the final query result to return to Charlie:

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

We note that in the query stage, no owner participation is needed to perform search.

Discussion

We can easily see that the above protocol satisfies both the search purpose and the privacy properties we defined in Section 2. Throughout both stages, users communicate only with the V-server, which store and return values. With keys encrypted under the K-server's public key, the V-server has no access to the keys, while the K-server has no information about the values or which user submitted the keys. Each rendezvous number is associated with a unique key-value pair so that the V-server can select the right values to return as the query results.

The storage overhead at each server 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. The search latency, however, will be slightly higher with public key encryption/decryption and one more round-trip communication between the V-server and the K-server. We will explore the protocol overhead in Section 6.

Values sometimes are just owner identities (e.g., on-line game player match). In such cases, we can reduce the storage overhead at the V-server by using a Bloom filter [2] to summarize the

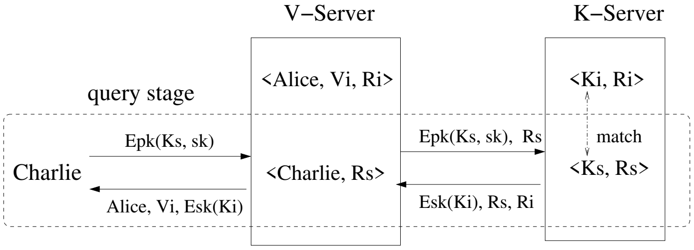


Figure 6: Supporting advanced queries (sk is a one time symmetric encryption key generated by Charlie.)

list of rendezvous numbers $R_1, \ldots, R_n$ corresponding to the key-value pairs of each owner. To reduce the storage overhead at the K-server, we can build an inverted index table [29] based on the keys.

During the data registration, rendezvous numbers can also be generated by owners based on data context to reduce the query result message size. For example, with keyword prefix match, an owner can select the rendezvous numbers such that the keywords with the same prefix (e.g., app*) will share the same digital prefix in the corresponding rendezvous numbers (e.g., 025*). Therefore, at the query stage, the K-server can summarize the query results as a rendezvous number range $[R_i, R_j]$ to reduce the result message size. However, in this case, the rendezvous numbers should be generated from a large space to avoid conflicts from different owners.

3.2 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 on the fourth floor 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 string “app*”.

To support returning 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 Fig. 6, 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.

4 Access Control and Authentication

In many applications, it is important for the owners to control which clients can search which data. For example, in stock quote dissemination, quotes should be searched only by paying customers. Moreover, without access control, a malicious user can carefully perform on-line dictionary attacks to slowly gather information about all the key-value pairs. Therefore, we need access control and user authentication mechanisms to prevent unauthorized client queries.

Since both the users and the servers may be located at different organizations and administrative domains (ASs), Peekaboo cannot assume a global name space. It should support queries of data from different owners seemlessly. More importantly, unlike traditional access control and authentication, the Peekaboo access control mechanisms should leak as little information about the data as possible in order to preserve user privacy. Finally, since search is a frequent operation to be performed daily, both the access control and authentication mechanisms should be convenient to the users. Supplying account names or passwords at every query is unacceptable to the clients. Our design is guided by the following principles:

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

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

3. *Convenience to the user.* Both the access control specification and the user authentication should be convenient. Owners should be able to revoke existing access permissions of their data easily. Clients should not need to know which data owners can potentially satisfy their queries prior to search.

With all users talking only to the V-server, one natural choice is to authenticate users at the V-server. In order to prevent the V-server from performing on-line dictionary attacks by granting access permissions to itself, we must enforce access control at the K-server. For inter-operability, the Peekaboo access control and user authentication mechanisms are based on public key cryptography and we assume an available public key infrastructure (e.g., [17]).

4.1 Access Control

A straightforward way to handle access control is to let each owner create an Access Control List (ACL) for a key value pair, specifying a list of clients that can access the pair. The owner then stores this ACL together with the corresponding keys at the K-server for permission checking. However, the client information in ACLs can be used to infer the corresponding key-value pairs. For example, an ACL with a list of New York clients might correspond to a key-value pair related with New York. Moreover, it is difficult to perform permission checking at the K-server without client identity information.

We propose a novel solution that hides client identities in ACLs while still enforcing access permission checking. The idea is to use client pseudonyms in the access control specification. The client pseudonym mappings are created by each data owner independently. They are kept secret to the Peekaboo servers, and released partially to the clients at the query stage for permission checking. We describe the scheme using the same example described in Section 3.1.

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:

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

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 with the K-server's public key pk so that only the K-server will be able to see the key and the access control specification. To help resolve the client-pseudonym mapping, Alice also encrypts the client pseudonym C_i' with C_i's public key pc_i. Finally, Alice submits the encryption $E_{pk}(K_i, ACL_i)$, the corresponding value V_i, and the client-pseudonym mappings $\{\langle C_1, E_{pc_1}(C_1') \rangle, \langle C_2, E_{pc_2}(C_2') \rangle, \ldots\}$ to the V-server:

$$\begin{aligned} \text{Alice} \rightarrow \text{V-server} \;&:\; E_{pk}(K_i, ACL_i), V_i, \\ & \quad \{\langle C_1, E_{pc_1}(C_1') \rangle, \langle C_2, E_{pc_2}(C_2') \rangle, \ldots\} \end{aligned}$$

Step 2*:* On reception of the registration request, the V-server registers the ownership, the value, and the client-pseudonym mappings locally:

$$\begin{aligned} \text{V-server} \;&:\; \langle \text{Alice}, V_i, R_i \rangle \\ & \quad \{\langle C_1, E_{pc_1}(C_1') \rangle, \langle C_2, E_{pc_2}(C_2') \rangle, \ldots\} \end{aligned}$$

The V-server then forwards the encryption $E_{pk}(K_i, ACL_i)$ as well as the newly generated rendezvous number R_i to the K-server:

$$\text{V-server} \rightarrow \text{K-server} : E_{pk}(K_i, ACL_i), R_i$$

Step 3*:* The K-server decrypts the message and registers the data and the access control information locally:

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

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 *Charlie*:

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

Step 2*:* Given the "ready-to-search" request from Charlie, the V-server extracts Charlie's pseudonym based on the client-pseudonym mapping $\langle Charlie, E_{pc}(C')\rangle$ if one exists, and presents the encrypted part to Charlie:

$$\text{V-server} \rightarrow \text{Charlie} \,:\, E_{pc}(C')$$

Step 3*:* Charlie decrypts the message and finds out his pseudonym 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, and sends the encryption back to the V-server:

$$\text{Charlie} \rightarrow \text{V-server} \,:\, E_{pk}(K_s, C')$$

Step 4*:* The V-server creates a rendezvous number R_s, and registers the entry $\langle Charlie, R_s\rangle$ for this query locally:

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

The V-server then forwards the encrypted message from Charlie to the K-server, attaching the rendezvous number R_s:

$$\text{V-server} \rightarrow \text{K-server} \,:\, E_{pk}(K_s, C'), R_s$$

Step 5*:* On reception of the query, the K-server decrypts the encrypted part, and gets both the queried key K_s and the corresponding client pseudonym C'. The K-server then performs both search and access permission checking. Only those query results that are allowed to be accessed by Charlie's pseudonym C' (e.g., R_i) will be returned to the V-server:

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

Step 6*:* Finally, the V-server looks up the values based on the K-server returned rendezvous numbers, and sends the query results back to Charlie:

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

Discussion

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' together with C_i's pseudonym C_i' using C_i's public key pc_i, and obtains the encryption $E_{pc_i}(C_i', G)$. Finally, Alice sends the client-pseudonym mapping with groups to the V-server during the data registration:

$$\text{Alice} \rightarrow \text{V-server} \; : \; \langle C_i, E_{pc_i}(C_i', G) \rangle$$

To revoke a client C_i's access rights on a particular key-value pair $\langle K_j, V_j \rangle$, Alice simply needs to remove 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 so that the V-server does not need to present encrypted pseudonyms to the client for decryption every time. An alternative solution is to let each client select a unique pseudonym and register it at different owners for permission specification. When submitting queries, a client simply attaches their pseudonym with the queried keys. However, with a unique pseudonym for each client, if two owners Alice and Bob both have granted permissions to the client Charlie, Bob could perform search, pretending to be Charlie by using Charlie's pseudonym, and easily find out all the key-value pairs from Alice that are search-able by Charlie to break Alice's privacy.

During the query stage, only clients can decrypt and get the corresponding client-pseudonym mappings. Thus a cheating client could modify them. To prevent pseudonym modification, owners can select pseudonyms from a large space (e.g., 128 bits) so that the K-server can easily detect a non-existing pseudonym to catch a misbehaving client.

4.2 User Authentication

For inter-operability, the Peekaboo user authentication is based on conventional digital signatures [9]. To defend against replay attacks, we use time-stamps and assume loosely synchronized clocks. When submitting a query to the V-server, the client Charlie generates a timestamp T, signs the timestamp T and the encrypted query $E_{pc}(K_s, C')$ with Charlie's private key, and sends the timestamp, the encrypted query, and the signature to the V-server in *Step* 3:

$$\text{Charlie} \rightarrow \text{V-server} \; : \; T, E_{pk}(K_s, C'), Sig(T, E_{pk}(K_s, C'))$$

On reception of the message, the V-server verifies the signature using Charlie's public key, which can be obtained from a public key infrastructure. The V-server then extracts the encrypted query $E_{pc}(K_s, C')$ from the message, and forwards it to the K-server to perform search using the process described above. For message integrity and confidentiality, we assume a protected channel such as TLS [27] between the K-server and the V-server.

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

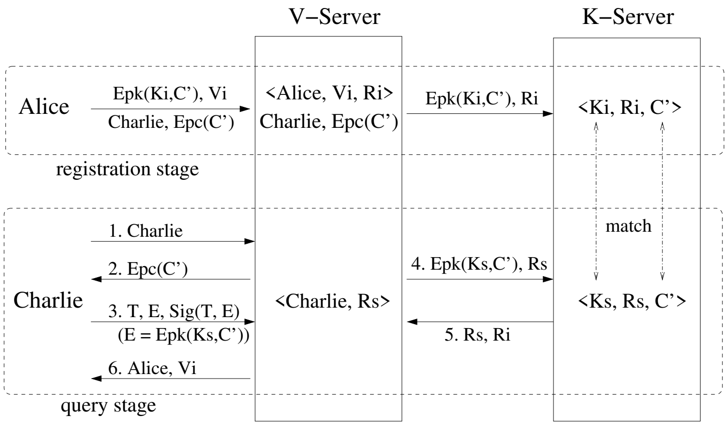


Figure 7: The revised Peekaboo search protocol with access control and user authentication

5 Deployment and Vulnerabilities

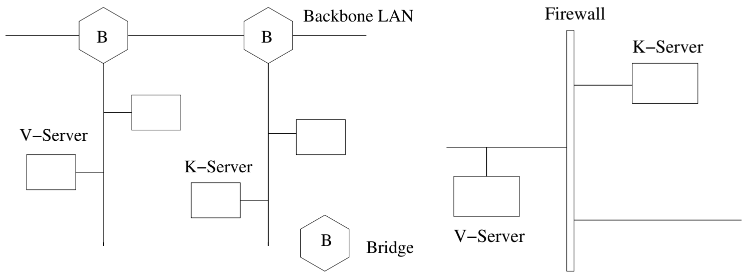
In this section, we discuss the various issues in system deployment. We also outline the types of malicious attacks that Peekaboo is vulnerable to and suggest possible solutions. Completely addressing these attacks is beyond the scope of this paper.

5.1 Deployment Issues

A basic assumption of the Peekaboo search protocol is that the K-server has no information about the user identity or the data values. Therefore, in a real deployment, the K-server should not have access to the network packets routed toward the V-server.

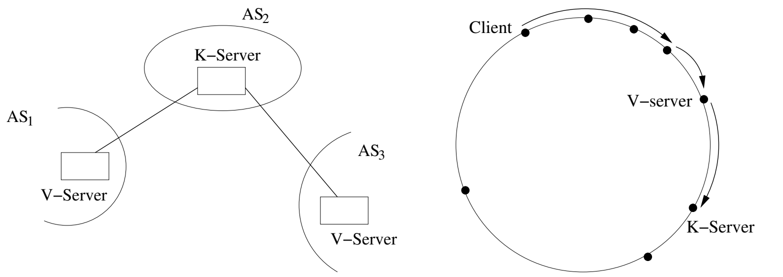
Local area networks. If both the K-server and the V-server are located on the same LAN, they should be configured at different network segments (e.g., separated by bridges). In a campus or company scale network with firewalls, we can configure the V-server to sit inside the firewall while the K-server can be configured outside the firewall so that it does not see the user traffic (Fig. 8).

Wide area networks. If both servers are on a wide area network, we require the K-server not be configured on transitive network backbones. Note that the K-server and the V-server can belong to different organizations or ASs. Therefore, each AS can deploy a V-server to accept user queries, while the K-server can be provided by a different AS (Fig. 9 (a)).



(a) Bridged LAN (b) With firewalls

Figure 8: Deploying Peekaboo in LAN



(a) WAN (b) Chord overlay networks

Figure 9: Deploying Peekaboo in WAN and overlay networks

Overlay networks. We are open to discussions of how to deploy Peekaboo on overlay networks, but here are some suggestions. In an overlay network such as Freenet [5], data and queries are routed incrementally in the application layer. Thus users can encrypt their identities (e.g., IP addresses) using the V-server's public key so that the K-server does not know who is the true message initiator. In a *Distributed Hash Table* (DHT) overlay network, for example, Chord [25], each host can serve as a V-server or a K-server. A key-value pair can be routed based on the hashing of the keys to locate the V-server, while the K-server can be the very next successor to the V-server on the Chord ring (Fig. 9 (b)).

5.2 Vulnerabilities and Possible Solutions

Both the K-server and the V-server could be active. For example, the V-server could produce bogus search results to the clients without forwarding the queries to the K-server. Similarly, the K-server could return arbitrary query results without performing the actual search. 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.

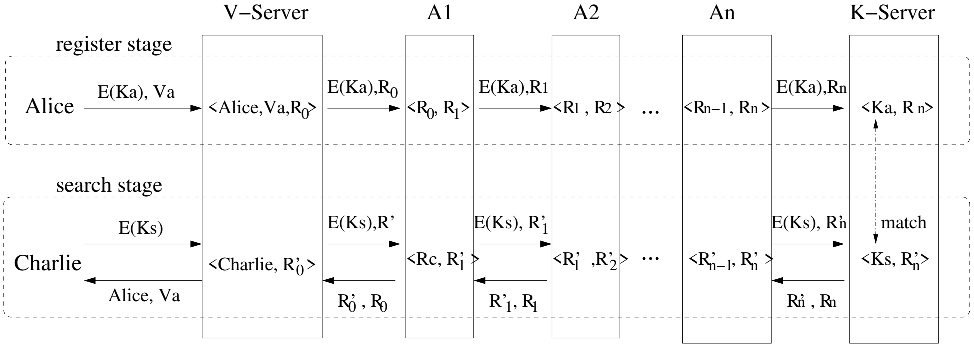


Figure 10: The Peekaboo multi-server search protocol

A more serious threat is server collusion, where the K-server and the V-server cooperate to reconstruct the key-value pairs by matching rendezvous numbers. One approach is to introduce further layers of indirection by adding auxiliary servers in the system. As illustrated in Fig. 10, all the servers connect with each other into a server chain. A registration request $\langle E_{pk}(K_a), V_a\rangle$ is first submitted to the V-server, who creates a rendezvous number R_0 and forwards $\langle E_{pk}(K_a), R_0\rangle$ 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 reserve direction. To tolerate the brute force collusion of up to $t-1$ servers, we need at least $t-2$ auxiliary servers between the K-server and the V-server.

However, the multi-server Peekaboo 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 the data and the queries. To mitigate such attacks, we can use solutions from [7, 26]. For example, each auxiliary server can buffer and reorder messages within a small time frame. In addition, auxiliary servers can set up fake clients to audit the K-server and the V-server by sending a batch of queries each time to detect if queries are hold and submitted from the V-server one by one.

Peekaboo servers could perform traffic pattern analysis to infer a particular key-value pair by measuring the frequency of the corresponding rendezvous number in query results. There are two approaches to mitigate such threat. First, owners can repeat the registration process to update the key-value pairs more frequently. Second, owners can purposely add noise in their data by allowing false positives in the query results.

With access control enforced by the K-server, a malicious K-server could submit queries as a client itself, returning all the rendezvous numbers of those query results that it is interested in. One solution is to let the V-server check if the K-server has returned a private data entry whose owner has not granted any permission to the client (i.e., there is no corresponding client-pseudonym mapping created by the owner). Meanwhile, the V-server can report the client query and the K-server returned query results to the corresponding owners based on random sampling to catch a misbehaving K-server. For stronger security against a malicious K-server, the V-server can trade off performance to check if the two ciphertexts about client pseudonyms (e.g., $E_{pc}(C')$ and $E_{pk}(C')$) correspond to the same plaintext (e.g., C') using techniques such as non-interactive zero-knowledge proofs [3], and to detect a misbehaving K-server or client.

Finally, like many other distributed systems, both the K-server and the V-server are vulnerable to various forms of denial-of-service attacks.

6 Example Applications and Performance

In this section, we describe an example application of file sharing such as Napster [20] 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 Napster-like file sharing system, owner store files or file names at servers. Each file has a owner-assigned local ID. Clients submit queries as keywords to the servers. If a file or file name 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.

Using the Peekaboo protocol, in the registration stage, owners register the file names (or file content) as the keys, and the local file IDs as the values at the K-server and the V-server, respectively. For each file, the V-server randomly generates a unique 128-bit string as the rendezvous number. To support efficient query search, the V-server computes a hash based inverted index of rendezvous numbers, while the K-server computes an inverted index table of keywords based on file names (or file content). 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.

To evaluate the Peekaboo protocol, we implemented both types of servers to support keyword search of file sharing. We use a Gnutella [12] trace gathered at CMU to conduct trace based experiments, and evaluate the system performance in the following three aspects: (1) the storage costs at both servers, (2) the search latency perceived by the client, and (3) the overhead of access control and user authentications. For comparison, we also implemented a regular centralized server that performs both data registration and queries, and repeat our experiments. All the servers are implemented in C++ in Linux, running on PIII 550MHz machines with 128 RAM in a 10BaseT Ethernet LAN. The public key encryption uses the RSA algorithm [22] 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++ library (version 4.2) [8]. Each data point in the figures below is the average of ten runs.

6.1 What are the Storage Costs at Both Servers?

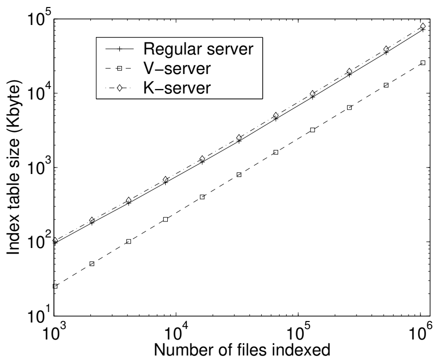


Figure 11: Index table size vs. number of indexed files

To evaluate the storage costs, we extract file names from the search reply messages in the Gnutella trace and register them using a fake owner program simulating different file owners who have submitted the reply messages in the trace. Fig. 11 shows the index table sizes of the Peekaboo servers and the regular centralized server as the function of the number of indexed files. We can observe that the storage costs increase linearly as the number of indexed files at the servers. 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.

6.2 What is the Search Latency Perceived by Clients?

In this section, we present the search latencies of the Peekaboo protocol. We implemented a client program running at a third machine (PIII 550MHz with 128 RAM) in the same Ethernet LAN. For each query, the servers return the first 100 matched files as query results. Fig. 12 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 Peekaboo protocol.

To further examine the search latency, we list the times spent in various steps of processing a query in Fig. 13. We fix the number of files indexed to be 100,000, and show both the

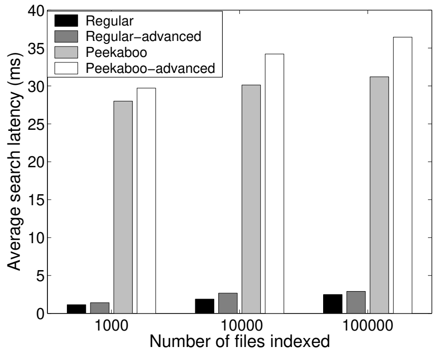


Figure 12: Peekaboo search latency

	Total	Network	Look up	RSA en.	RSA de.	AES en.	AES de.	Other
Mean	36475	6427	3041	1575	23834	581	781	236
Std dev	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 13: Time to process a search request using 1024-bit RSA keys and 128-bit AES keys (μs).

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 in plain text. During the query processing, 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 the clients since the network latencies on WAN are usually or the order of tens of milliseconds. 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.

6.3 What is the Overhead of Access control and Authentication?

The Peekaboo access control and user authentication mechanisms introduce the following extra steps during the query processing: (1) client signature signing and verification, and (2) client pseudonym encryption and decryption. 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.

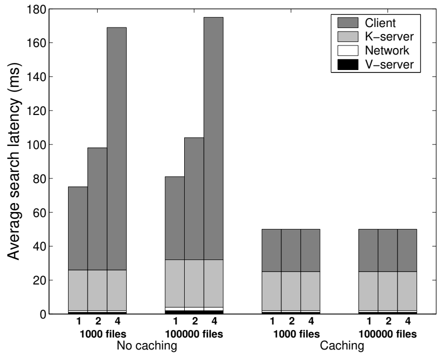


Figure 14: Search latency with access control and user authentication. The client is associated with 1, 2, 4 pseudonyms, respectively

Fig. 14 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 due to the expensive RSA decryptions. 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.

7 Related Work

Related work comes from four areas: searching over encrypted data, anonymous communications, private information retrieval, and multi-party computation.

Song, Wagner, and Perrig have proposed a cryptographic scheme for search on encrypted data [23] at untrusted servers. In their scheme, data is stored in encrypted forms at servers.

A client query is also encrypted and the servers perform search by sequentially checking if the query string and the encrypted data strings follow the same patterns. Since both data and queries are encrypted, their schemes require the clients to share the same encryption keys used by the data owner in order to perform search, limiting the search to be performed by highly trusted clients. In addition, the number of cryptographic operations grow linearly with the document lengths, limiting the amount of data to be stored and searched.

There has been a large body of literature on anonymous communications to prevent discovery of source-destination patterns. In general, there are two types of approaches for achieving user anonymity: proxy based approaches and mix based approaches. In the proxy based approaches, the system interposes an additional proxy between the sender and the receiver to hide the sender's identity from the receiver. Examples include email pseudonym servers [19], Janus [15], and Crowds [21]. The Peekaboo V-server bears some similarity with a proxy in that all user traffic goes through it. However, the primary purpose of the Peekaboo protocol is not to hide user identities, but rather to perform search without revealing the key-value pairs. Thus the Peekaboo V-server is not only a proxy as it actively participates in storing and returning values. In the mix based approaches (e.g., [7, 26]), a chain of proxies are interposed between the sender and the receiver to achieve unlinkability between the sender and the receiver. We showed in Section 5.2 where we used mix based approaches to prevent timing attacks in the multi-server Peekaboo protocol. Compared with these approaches, Peekaboo protects key-value pairs as well as user identities. Our two-server protocol implementation is much more lightweight and thus more practical to use. However, Peekaboo does not provide unlinkability between key-value pairs in the presence of server collusion.

The problem of Private Information Retrieval (PIR) [4, 11] has been well studied so that clients can access entries in distributed databases without revealing which entries they are interested in. These works model the database as an n-bit string, and the user retrieves the i-th bit, while servers gain no information about the index i. Although PIR schemes can achieve very strong security, they are generally not practical to use. In contrast, Peekaboo protects both the owner privacy and client privacy in more general key-value search systems. It requires only 2 servers and is efficient in search performance.

Secure multi-party computation (SMC) is also intensely studied [13] where data D are divided and stored at n servers, which jointly perform a computation to reconstruct D while no $n-1$ servers can compute D. Key-value search can be viewed as a special type of multi-party computation. However, SMC solutions usually have high computation and communication complexity, and thus are not efficient enough for practical use.

8 Conclusion

In this paper, we have proposed a protocol called *Peekaboo*, for performing general key-value search at untrusted servers without loss of user privacy. Specifically, 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. Since key-value pairs release information only if they are together, our main idea is to separate the key-value pairs and store them on different servers. Supported by access control and user authentication, Peekaboo is: (1) secure in that search can be performed only by authorized clients while protecting the privacy of both the data owners and the 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 practical to use today.

References

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

[2] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commnications of the ACM*, 13(7):422–426.

[3] M. Blum, A.D. Santis, S. Micali, and G Persiano. Non-interactive zero knowledge. *SIAM Journal of Computing*, 20(6):1084–1118, November 1991.

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

[5] 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*.

[6] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. An architecture for a secure service discovery service. In *Mobile Computing and Networking*, pages 24–35, 1999.

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

[8] Wei Dai. Crypto++.
http://www.eskimo.com/˜weidai/cryptlib.html.

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

[10] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste. Project Aura: Towards distraction-free pervasive computing. In *IEEE Pervasive Computing 1*, pages 22–31, 2002.

[11] 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.

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

[13] O. Goldreich. Secure multi-party computation. First version posted in June 1998. Final revision posted October 2002.

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

[15] The Lucent Personalized Web Assistant. http://www.bell-labs.com/project/lpwa/history.html.

[16] 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*, November.

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

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

[19] D. Mazieres and M. F. Kaashoek. The design and operation of an e-mail pseudonym server. In *5th ACM Conference on Computer and Communications Security*, 1998.

[20] Napster. http://www.napster.com.

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

[22] 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), February 1978.

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

[24] M. Spreitzer and M. Theimer. Providing location information in a ubiquitious computing environment. In *Proceedings of SIGOPS'93*, pages 270–283, 1993.

[25] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of ACM Sigcomm*, August 2001.

[26] P.F. Syverson, D.M. Goldschlag, and M.G. Reed. Anonymous connections and onion routing. In *Proceedings of the 1997 IEEE symposium on Security and Privacy*, 1997.

[27] The TLS Protocol. http://www.ietf.org/rfc/rfc2246.txt.

[28] M. van Steen, F. Hauck, P. Homburg, and A. Tanenbaum. Locating Objects in Wide Area Systems. In *IEEE Communications Magazine*, pages 104–109, 1998.

[29] I.H. Witten, A. Moffat, and T.C. Bell. Managing gigabytes: Compressing and indexing documents and images. *Second ed. Morgan Kaufmann*, 1999.