PIRpeer: Distributed, Oblivious Keyword Search

Alvin AuYoung, Barath Raghavan, Chris Stotts, Erik Vandekieft
University of California, San Diego
Final Project Report, CSE291D, Spring 2003

Abstract

We examine the problem of oblivious keyword searching in a distributed setting. In this problem, a client contacts a server or set of distributed servers and makes a request for a piece of information, but without revealing to the server what information is sought. Nevertheless, the server returns the correct answer. This problem, called the *Private Information Retrieval* (PIR) problem, has been extensively studied in the cryptography literature. We construct a real, distributed system, `PIRpeer`, comprised of search servers built upon PIR primitives to achieve practical keyword searching in the context of keyword searches for peer-to-peer distributed hash tables (DHTs) or the like. In particular, our system, combined with existing anonymity routing layers and strong-data anonymity DHT-like file storage systems, provides the last piece in an end-to-end anonymous lookup and retrieval system.

1 Introduction

As corporations increase their efforts to invade the privacy of individuals, users perceive a greater privacy threat from both the legal system and large corporations. This has led many in the security community to develop large-scale systems to provide both anonymous storage [10, 11] and anonymous routing [7, 12, 14]. However, we contend that one critical piece of the security puzzle is missing – when using existing anonymity schemes, a user's identity is not revealed, but the content sought is revealed both to the server and to any eavesdropper. Although it may seem that by protecting a user's identity, an anonymity scheme protects the user's privacy completely, in fact, the content requested may be highly correlated with a particular user. For example, suppose a public gene database were run by an HMO – a user with a particular rare genetic disease could attempt to search for a sequence of their own DNA to read about latest advancements, but would like to do so without revealing to the HMO the sequence, since it can inextricably tie the *query* to the user, despite any anonymity scheme in use. Since it is impossible, even with collusion among servers, for anyone to determine the contents of a client's request, the guarantees that we can provide are much stronger than those provided by most anonymity schemes, as they cannot resist complete collusion.

Since some storage schemes provide weak-server anonymity, the server must go to some effort to determine the data sought by a client. However, these systems provide no efficient means for search. A naive implementation of search for such a system would likely completely reveal the user's search queries even to an unmotivated server. There exist widely studied theoretical solutions to this problem of Private Information Retrieval in the cryptographic literature, but to our knowledge, no practical systems apply such techniques to a search service or the like. Our system, `PIRpeer`, provides oblivious keyword search across a series of replicated, distributed search servers. We intend `PIRpeer` to be compatible with existing storage and routing anonymity systems to achieve a complete, end-to-end secure search, route, and retrieval system.

2 Private Information Retrieval

We define the problem of Private Information Retrieval in the context of keyword search, variations on the problem we describe, and possible common relaxations.

2.1 Problem statement

Our system uses a two-party protocol that solves the Private Information Retrieval (PIR) problem with a client A and a server B in which B has a sequence of strings $s = (s_1, \ldots, s_n)$, each of length l bits. A wants to receive a particular $s_i, 1 \leq i \leq n$ from B without revealing i to B. To accomplish this, A sends a request to B and B responds with the data[1]. We do *not* require that A only learn s_i – A may learn information about other strings as well. In the context of keyword search, we say that a client retrieves a listing of keywords $k = (k_1, \ldots, k_n)$ that correspond to the strings in s, and chooses i based upon the keyword sought.

In a more practical formulation, we expect that a solution to this problem achieves communication complexity (in practice, if not asymptotically) much smaller than a trivial scheme in which s itself is sent. Our design is informed by the assumption that n is large and l may be large.

[1] While there is no requirement that the protocol be a single-round protocol, for simplicity we only deal with single-round versions

As is clear from the problem statement, we are only interested in client privacy, not server privacy. A related problem statement includes that A may not learn about any s_j for $j \neq i$. This is known as the Oblivious Transfer (OT) problem, as we describe later.

3 System Architecture Overview

The `PIRpeer` system consists of a potentially large number of servers with an even larger number of clients. These servers act as search servers and respond to client queries for particular keywords. Both clients and servers utilize appropriate secure protocols to assure client privacy. Clients potentially communicate with servers for secure, distributed file storage systems, once a keyword query has been answered, to retrieve actual data. In addition, all communication channels may utilize a secure, anonymous routing/tunneling protocol to provide end-host privacy.

To provide our search service in a wide-area context, we aim to make the search service both distributed and replicated. Our system is comprised of a set of search servers grouped into a *replica group*. Each replica group is responsible for replying to queries for a certain part of the keyword space. We ensure that keyword entries are not lost. Our server infrastructure replicates and reorganizes such that for some r_{min}, we ensure that data is lost only if r_{min} servers fail simultaneously.

In `PIRpeer`, different replica groups are responsible for different parts of the keyword space. This distribution is achieved by using a self-adjusting binary trie on hashes of keyword strings in which leaf nodes represent replica groups. Each keyword is hashed using a pseudorandom hash function and placed in the appropriate replica group. Our system ensures that no fewer than some constant r_{min} servers are in any group, thus assuring that the system is resilient to server failure.

Within a replica group, servers ensure that all updates made to any individual server within that group (the addition of keywords into the system) are only committed by all servers or none servers using existing atomic broadcast protocols. In our design, either a centralized or hierarchical view management system can be used; we implement the first and describe how the design of the second can be implemented as a drop-in replacement.

4 Server Design

The keyword database maintained by the PIRpeer system is *not* the same at every server. Each server in PIRpeer decides which part of the database it will store, based on the afore-mentioned pseudorandom hash function. Each server stores a copy of the *directory tree* which is kept the same at every server using consistent replication. We describe the procedures for replication in more detail in the following sections.

4.1 Database Maintenance

Given the keyword hash tree, a server knows in which leaf node it belongs, which data it should store, and to whom it must communicate with retrieve data. We maintain servers in *replica groups* which are leaf nodes in the keyword tree with multiple replicated servers in them. These replica groups always remain consistent, even with tree splits and merges, server joins and leaves, and node failures. We ensure that each replica group remains consistent by transferring keyword additions as the protocol progresses and sending data appropriately when the tree splits or merges.

4.1.1 Replica Group Maintenance

We considered several alternatives to our current group maintenance scheme. One seemingly trivial solution to maintenance of multiple nodes in a replica group is to use a protocol such as `rsync` breaks under network partition or simulated network partition. Such protocols are leader based, and as such, if a few nodes believe the leader has failed, then it is possible for them to elect a new leader deterministically, but artificially partition themselves by running a separate network segment.

Next, we considered the use of vector timestamps to ensure ordering on messages to a replica group. But vector timestamps only ensure a consistent ordering among the sites that receive a given update; if someone were to die while in the process of sending an update to a group, members of a group would become out of sync. Also, without guaranteed message delivery in bounded time, vector timestamps become difficult to manage. For practical purposes, while using TCP, one could consider the time limit for message delivery to be 120 seconds, as that is the TCP timeout on most implementations, but this muddles the issue of failure detection with message delivery.

A very heavyweight solution to the problem, using viewstamped replication much like Harp [17] would solve the problems we need solved, but with extremely high overhead. We instead wanted an easy to implement and deploy solution that would still provide replication in the face of failure. The work by Birman and Joseph describes a lightweight inter-site communication layer, consisting of a a view management protocol with a view manager and an array of hosts. They maintain a view sequence, and when servers are added and deleted, proposed view extensions allow servers to notify others of the possible change in view. We use this view to know to whom updates are propagated. However, we do not implement ABCAST itself within replica groups – we had intended to do this, but had insufficient time.

4.1.2 Handling Tree Changes

Our servers maintain a constant degree of replication at all times by use of a self-adjusting binary trie to or-

der servers and the keyword space mapped to them. We construct a binary trie in which all replica groups exist as leaf nodes in the trie, with two constants r_{min} and r_{max}. The trie represents an organization of hash values of both keywords and server identifiers. We have overloaded the notion of this trie in the sense that both server lookups (which happen on the hash of the server's hostname and port) and the keyword lookups (which happen on the hash of the keyword strings) are performed by traversal of the trie.

The structure of our trie provides several important benefits to our server replication. First, when a split occurs, a server can simply prune half of its keyspace, as after a split its database must contain only a subset of its prior data. Second, upon a merge operation, a server never has to remove the data it currently stores – instead, it only has to append the memory of sibling replica groups. Third, because we construct the trie based on a pseudorandom hash function (in our case, SHA-1), we are guaranteed that the trie will remain statistically balanced.

Figure 1 shows a directory tree (the trie) in which servers have been stored at leaf nodes. For example, A has a hash that begins with 00, as does B, C's hash starts with 01, and so on. In this example, $r_{min} = 2$. Suppose server F were to fail. Figure 2 shows the directory tree after the removal of F. Note that since the tree would have had a replica group with only 1 server, which would have gone below r_{min}, the replica group merged and caused a recursive merge on the other branch, producing a trivial tree that maintains our replication property. A replica group splits only when it contains more than r_{max} servers and the splitting operation does not cause any replica group to contain fewer than r_{min} servers.

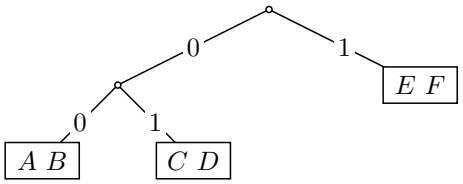

Figure 1: A sample directory tree

Figure 2: The directory "tree" after removal of F

4.2 View Management

Servers must maintain a consistent view of the directory tree so that clients will know which replica group they need to communicate with in order to perform a search, and also so that our scheme of merging and splitting will not lead to data loss or inconsistency.

4.2.1 Protocol

To maintain a consistent site view, all servers in our network implement a site view management protocol that was first described by Birman and Joseph [5]. The protocol ensures that every live server will see the same set of server additions and removals, and that the ordering of these additions and removals will be consistent at every server.

The protocol relies on a *view manager* to coordinate a network-wide two-phase commit of any *view extension*, where a view extension is some set of *view changes*. A view change is simply a notification about the addition or removal of a server. The number of view changes in a view extension is usually one, unless several servers die or attempt to join the network simultaneously (to be precise, whenever other servers die, or some new servers are able to get a join request to the view manager before the view manager is able to complete its two phase commit for some previous extension). When this happens, the new failure or addition is appended to the old one and the view manager begins the whole two-phase commit process over again for the new, elongated extension.

The protocol is robust to arbitrary numbers and orderings of failures. For example, if the view manager fails while in the process of sending out commits for a view extension, the next view manager will realize that the old view manager has failed (in our implementation, failures are always detected by either a failed `write()` or the receipt of `EOF` from a socket), and will create a new view extension containing the failure of the old view manager and append that to either the old uncommitted view extension, if the new view manager had not yet received its commit, or the last committed extension, if it had received its commit. Servers will send a *positive ack* to a manager if the view extension they have been sent does not contain any missing view changes; they will send a *negative ack* and a list of the missing view changes if the new extension is incomplete. (This happens whenever a view manager sends out an extension to part of the network, dies before sending to everyone, and the new view manager was one of the nodes which did *not* receive the extension.)

4.2.2 Benefits

Keeping the different servers' perceptions of the ordering of server additions and removals is critical to our implementation because every server maintains its own

copy of the directory tree. Operations on the directory tree must be done in the same order or else the trees will begin to diverge. It is critical that this not occur because the structure of the directory tree is what servers use to decide what part of the keyword space they are responsible for; if the servers in the network were to disagree over who needs to cover what part of the keyspace, sections of the database could potentially be lost in the resulting confusion.

Therefore, our site view management protocol serves two main purposes. First, it keeps the servers' directory trees consistent, and second, since it is a form of two-phase commit, it ensures that it is impossible for a new commit phase to begin before the previous has been completed. This latter property is critical to the correctness of our merging and splitting scheme. It is upon receipt of a commit that all merging and splitting, and the resulting shuttling around of data, occurs.

4.2.3 Procedure

The typical sequence of events is as follows. One server, the very first server in the entire network, starts up and is explicitly designated as the "root" on the command line. The root initializes a blank directory tree, adds itself to it, and commits it, then enters its select loop, awaiting incoming connections. All subsequent servers must specify some other existing server to connect to on the command line, which they connect to and send a "join request". If the server that receives the join request is not the current manager, it sends a "join redirect" with the manager's IP address and port. If the manager receives a join request, it serializes its directory tree, sends it to the requester, and initiates a site view extension containing the addition of the requesting server. Before acking a new server addition, each server will establish a TCP connection to the new server (or will ensure that a connection already exists, if for example the new server had already established a connection to a server in order to send it a join request, and was then redirected.)

4.3 Client-Server Communication

Clients perform add and search operations on the servers. No assumptions are made by the clients about the lifetime of the individual servers or specifics about the network topology. The client-server protocol apportions responsibility for maintaining robustness and data consistency equally between the client and server.

5 Oblivious Retrieval

At the heart of our system is a protocol that solves our problem statement, namely, one that achieves client privacy of information retrieval. We have studied many of the existing approaches and protocols and offer observations and descriptions of them and their relevance to our goals.

5.1 Trivial Transfer

In a trivial implementation of the protocol, the client A makes a request to the server B asking for s in its entirety. Doing this clearly reveals nothing to B of which s_i is sought. However, the communication complexity of such a scheme is $O(n \cdot l)$, which we contend is the worst case performance of any such protocol that aims to achieve client privacy. However, its advantage is also significant – the computational cost of such a trivial transfer is negligible.

5.2 Oblivious Transfer

A very strong, expensive scheme is Oblivious Transfer (OT). As we described before, we aim to only solve the problem of client privacy in the information retrieval process. However, much past cryptographic work has been done on OT protocols [4, 21] that protect the server as well and present either a 1 out of 2 OT protocol [20] or a generalized 1 out of n OT [19]. In such a scenario, the server knows nothing about which string was returned to the client and the client knows nothing about the strings she did not receive. Due to the strong guarantees of OT protocols, they have high communication and computational complexity, and thus we do not use oblivious transfer.

5.3 Searching Encrypted Data

A slightly unrelated, but alternative approach to this problem is based upon the searching encrypted data protocols of Song *et al.* [25]. Their solution provides a technique for a client A to place encrypted data on a server B and have B efficiently search the data for some string without revealing to B a key that would enable decryption of any of the data. However, there is a significant problem with this design: arbitrary clients must be capable of searching this encrypted data at any server, so they must be capable of retrieving the key. If a client can request a key for the data at a server, the server, by setting up its own client instance, can do the same, revealing in effect any subsequent client queries that pass through the server.

We considered many approaches to fixing this problem, including the use of a secret sharing scheme [18, 23] to share the key with some level of verification that a client is not associated with a server. Without assuming away collusion, the only viable option with this searching protocol is to invalidate the entire database at a server every time a search is performed and replace it with a randomly shuffled replacement of the same database. Due to its high communication overhead, we do not use this scheme, but note that it holds great promise if the key distribution problem were solved.

5.4 Secure Multi-Party Computation

Another cryptography technique, secure multi-party computation, has been studied extensively for the purposes of voting protocols [13]. In such a protocol, a function f with multiple inputs $i_0, i_1, \ldots, i_n$ is computed in a distributed fashion among n parties such that each party k provides one input i_k to f and no party knows another party's input to f. Most protocols for this problem, due to their generality, have steep computational and communication requirements. However, should such overhead be reduced, this technique holds promise as it allows arbitrary execution of some function which may aid in more complex client-server interactions for searching in a secure setting.

5.5 Private Information Retrieval

The private information retrieval problem [9] is exactly the problem we aim to solve. PIR protocols provide client privacy and aim to reduce communication complexity over other forms of client/server privacy, such as OT. In the strongest form of PIR, information-theoretic PIR, the server B obtains absolutely no information (in an information-theoretic sense) about i from the client A's query. The information-theoretic PIR problem was originally examined in a two server model in which the two responding servers were assumed to not communicate with one another yet have replicated databases. Later work extended this to a protocol with k servers with reduced complexity [1]. Other techniques argue for server-side preprocessing of data to reduce the response time [3]. While keyword searching has been studied with PIR [8], the existing schemes do not address single-server PIR and assume that servers do not collude.

However, the information-theoretic guarantees in early PIR schemes were much stronger than the usual, computational hardness guarantees associated with most cryptographic primitives. An unfortunate side effect of this was that no single server PIR scheme with sublinear communication complexity existed. Kushilevitz broke this barrier by relaxing the problem into the Computationally-Private Information Retrieval (cPIR) problem [16], in which the query is secure under standard cryptographic hardness assumptions, but is not information-theoretically secure. Further work [6, 15]has reduced this communication complexity to polylogarithmic levels.

One critical, and often overlooked, problem with existing single-server PIR schemes is the *computational* complexity. Indeed, as several researchers discovered [24], to use a PIR scheme on a large database even with auxiliary processors was practically infeasible. However, due to our architecture, we ease these problems in several ways:

- Only the hashes of files are stored in our database, not the files themselves
- The keyword hash space is subdivided into many different replica groups, and as such, require computation and communication for a smaller database
- Since replica groups can have potentially many servers, PIR requests can be easily served in parallel over the same data set

In our system, we use the protocol proposed by Yang [26] that relies upon the Quadratic Residuosity Assumption (QRA), a standard cryptographic hardness assumption.[2]

5.6 Repudiative Information Retrieval

An extension to PIR was proposed by Asonov [2] in which the guarantees of the protocol are only that a server cannot tell *for sure* which value the client requested, but has a non-negligible probability of determining which better than random guessing. In his scheme, the security of the protocol is dependent on a parameter which can be set such that the scheme provides as much security as PIR (albeit slower), and at the other extreme provides no security. However, his protocol seemingly requires a secure co-processor on the machine to ensure security, and as such, is infeasible for our system, in which servers may not have any specific type of hardware.

6 Client Design

A `PIRpeer` client performs both searches and additions of data on behalf of the end-user. In order to achieve its privacy guarantees, `PIRpeer` must complement a secure communication protocol with a secure storage service. To this end, a `PIRpeer` client is designed to interface with a secure storage service in addition to secure search servers.

6.1 Theory of Operation

A user performs a search by specifying a set of keywords to search for. Each data item stored in the storage servers is associated with one or more of these keywords. These data items are represented as a hash value in the search servers. The result of a user query (search) is a set of hash values corresponding to the data items that are most strongly associated with the set of keywords in the user's query.

Data items and their corresponding hashes are added to the system by the client on behalf of the end user. The client performs separate operations to store the data

[2]His protocol represents a straightforward extension of an existing protocol also based upon QRA. To our knowledge, his protocol was never formally published, likely because its contribution does not reduce asymptotic complexity, but does turn out to be useful for our purposes.

item on the storage server and to store the corresponding hash value on the search server.

6.2 Search Server Interface

The knowledge of which particular data items are available in the system is organized in a database that is distributed across the `PIRPeer` search servers. The database roughly corresponds to a mapping between keywords and a set of hashes corresponding to data items that are associated with the keyword. Data is distributed in the system based on the hash value being searched. Thus the topology of the network determines where the hash values are stored. The client requests a directory tree object from a search server in order to determine the network topology, and which server to query for particular hash values.

6.3 Storage Server Interface

The client interacts with the storage server to either store or retrieve data. We assume a storage server interface similar to that of Freenet. To add data to the system, a user specifies which data to add, and the storage server will generate a hash value corresponding to this data. The client generates a set of keywords to describe the data (in the current implementation, the user specifies which keywords to associate with this piece of data) and sends this information to a particular search server, which then adds this information to the database.

6.4 Server Discovery

At startup, a client is assumed to have the identity (hostname and port number) of one search server and storage server. It is assumed that there are one or more well-known search servers for clients to bootstrap clients.

Upon retrieval of a directory tree from the server, a client can choose to store as many server hostname/port pairs as it wishes to, and it can continually refresh this list with subsequent directory tree retrievals. This will be necessary as search servers may join and leave the network at any time.

Currently, the client maintains a list of servers each of which it can query for the directory tree. Upon server disconnection, the client chooses another server at random to try to establish a TCP connection to. The number of retries is a tunable parameter as is the frequency of serverlist refresh.

6.5 Fault Tolerance

Clients maintain state for each user request until it is satisfied by a server. Because client failures, departures, or network topology shifts can occur at anytime, a client request may fail at anytime. To move unnecessary complexity out of the server consistency protocols, the clients maintain state for each of its requests, and will retry it until a server acknowledges executing the particular task.

Furthermore, the client can proactively fetch updated directory trees in order to maintain an up-to-date list of servers currently on the network. In fact, it is necessary that long-lived clients perform periodic updates in order to stay connected to the system.

7 Layers/Related Architectures

Our system is designed to be a component in an end-to-end client privacy solution. We describe several other complementary systems and how `PIRpeer` relates.

7.1 Storage Systems

First, since `PIRpeer` allows for keyword searching on a secure DHT-like file storage system, we describe several systems for which it would use after the keyword search is complete. Freenet and free haven are systems for anonymous, distributed storage. They provide passive-server document anonymity, meaning that a server must actively attempt to determine the data it stores should it want to know. This property allows for plausible deniability in the case of legal attacks, but does nothing to prevent malicious servers from knowing what data they store, and thus, what data a client is retrieving. Since we argue that our system provides guarantees for something stronger in terms of keyword searching, but such a guarantee is useless in the event that the client must retrieve the file from a system with only passive-server anonymity. An alternative is a system (and likely more like it to come) that provides active-server document anonymity [22].

7.2 Routing Systems

In the space of routing systems that provide guarantees of anonymity, common examples are variations on Chaumian mixes [7]. Onion routing [14], a system developed by the US Navy, allows for layered encryption and decryption of packets as they are routed through a network of forwarding hosts. Barring large scale collusion, the system allows for anonymous communication through these forwarding hosts. A newer solution, Tarzan [12], provides a fully distributed, peer-to-peer anonymizing layer that provides an IP abstraction to hosts. The system does not rely upon a core set of mixes, and therefore is more robust to attacks.

We envision that clients could use such an anonymizing layer to communicate with `PIRpeer` search servers. This would provide a high degree of end-host anonymity, keyword search anonymity, and data retrieval anonymity, providing the end-to-end solution we described.

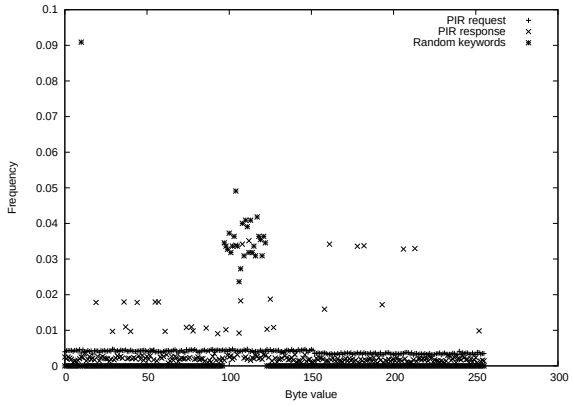

Figure 3: Distributions of bytes for keywords, PIR request, and PIR response.

8 Possible Attacks and Remedies

We acknowledge that there exists at least one style of attack against the client privacy our system provides. A `PIRpeer` server and a colluding Freenet (or other storage system) server with only passive-server document anonymity could possibly attack our system in the following way: The server could add a large number of fake file entries with keywords it is interested in. Once a client retrieves these files, the server could determine which files the client was interested in, and thus, which keywords were likely requested.

We contend that attacks such as the one described herein are not attacks against `PIRpeer`, but rather, against `PIRpeer` with a file storage system. If a system were to provide active-server document anonymity, then a server would be unable to determine whether certain data items were indeed stored in particular locations.

9 Analysis

To provide a more empirical evaluation of the randomness of the output of our PIR-based protocol, we provide a plot in Figure 3 of the byte value distributions for both the PIR request and PIR response from the server. Also shown in the graph, for reference, is a distribution on randomly selected strings (to represent keywords). Note that the PIR request from the client is uniformly distributed, and that the PIR response is almost uniformly distributed (but since it is smaller, suffers from a greater degree of noise). This is in marked contrast to the keyword string byte distribution. While heuristic analysis of data sets is rarely considered for cryptographic protocols, we consider it here merely because it sometimes reveals data patterns that are assumed to be invisible by cryptographic hardness assumptions.

Among our experiments and testing, we tried various distributions of servers among the keyspace. In particular, one of the more interesting cases involved 12 servers distributed such that one replica group was at depth 1, one was at depth 2, and two were at depth 3. Through quick successive terminations of servers in the highest level group, we were able to force a large three-level merge and check for consistency. Through such merges, and multiple view changes, the servers maintained consistency.

10 Future Work

We leave many aspects of this project and space as future work. First, there exist many optimizations we have examined but not implemented with respect to view and tree management. Furthermore, there are possibly techniques to reduce the chance of collusion in a k-server PIR scheme with a threshold cryptography scheme to assure that the system can withstand a bounded amount of collusion. This would allow us to possibly utilize a more efficient multi-server PIR scheme. Also, to improve performance, we describe possible work on creating fixed block sizes in the database. Finally, we describe our need for a multi-server, threshold cPIR scheme.

10.1 View/Tree Management

We considered various designs of our view and tree management systems that involve using a decentralized tree. Such a change would help the system scale as the number of search servers increased, since servers would not require knowledge of the entire directory tree. Our proposed hierarchical update scheme would use a directory tree structure much like the one we described, except that all internal nodes would have a virtual "leader" server chosen from one of the servers in a child group below that internal node. Each internal node would only be responsible for knowing the servers beneath it and above it. Each leaf would only need to know their parent node. Upon search, at most $O(\lg n)$ recursive directory lookups would have to occur to find the appropriate replica group, and while this is more than the current system, it would provide better scaling properties, especially due to server failure. Also, using a technique similar to path compression in tree-based disjoint set structures, servers could cache pointers up to higher levels and vice versa, saving several traversal steps upon each subsequent lookup.

10.2 Database Management

We considered a performance enhancement for our database in conjunction with PIR. In the case that the number of file hashes per keyword follows a highly skewed distribution, in our current design, each block is padded to the length of the maximum length keyword entry for the purposes of PIR. However, this wastes a great deal of space. Instead, we propose to modify this such that large keywords (those with many corresponding hash values) are broken once they reach a maximum

size into multiple fixed size chunks. These chunks are sorted. When a client makes a request, not only is PIR more efficient since less space is wasted due to padding, but also the client may perform a binary search on the numbered keyword chunks to perform an intersection with known file hashes for a multi-keyword query.

10.3 Threshold cPIR

In the context of PIR, many multi-server information-theoretic PIR schemes have been presented, but no multi-server cPIR schemes have been proposed to our knowledge. We contend that one way to reduce the high cost of cPIR is to distribute the work across multiple servers. In the case of PIR, distribution across multiple, non-colluding servers caused a super-linear speedup in the communication complexity of the protocol. We expect similar results can be gained for cPIR. However, a major drawback to multi-server PIR schemes in general is that collusion is assumed to be impossible, whereas, especially in a scheme like ours, collusion is likely easy. Instead, we hope to develop a threshold cPIR scheme that resists a fixed, small amount of collusion among replicated servers.

References

[1] A. Ambainis. Upper bound on communication complexity of private information retrieval. In *ICALP*, 1997.

[2] D. Asonov and J.-C. Freytag. Repudiative information retrieval. In *Proceeding of the ACM workshop on Privacy in the Electronic Society*, pages 32–40. ACM Press, 2002.

[3] A. Beimel, Y. Ishai, and T. Malkin. Reducing the server's computation is private information retrieval: Pir with preprocessing. *Lecture Notes in Computer Science*, 1880, 2000.

[4] M. Bellare and S. Micali. Non-interactive oblivious transfer and applications. In *Proceedings of Crypto'89*, 1989.

[5] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems (TOCS)*, 5(1):47–76, 1987.

[6] C. Cachin, S. Micali, and M. Stadler. Computationally private information retrieval with polylogarithmic communication. *Lecture Notes in Computer Science*, 1592, 1999.

[7] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[8] B. Chor, N. Gilboa, and M. Naor. Private information retrieval by keywords. In *TR CS0917, Department of Computer Science, Technion, Israel*, 1997.

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

[10] I. Clarke, T. W. Hong, S. G. Miller, O. Sandberg, and B. Wiley. Protecting free expression online with freenet. *IEEE Internet Computing*, 6(1):40–49, 2002.

[11] R. Dingledine, M. J. Freedman, and D. Molnar. The free haven project: Distributed anonymous storage service. *Lecture Notes in Computer Science*, 2009, 2001.

[12] M. J. Freedman and R. Morris. Tarzan: a peer-to-peer anonymizing network layer. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 193–206. ACM Press, 2002.

[13] O. Goldreich. Secure multi-party computation. Working Draft, 2000.

[14] D. Goldschlag, M. Reed, and P. Syverson. Onion routing for anonymous and private internet connections. *Communications of the ACM*, 42(2):39–41, 1999.

[15] A. Kiayias and M. Yung. Secure games with polynomial expressions. *Lecture Notes in Computer Science*, 2076, 2001.

[16] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings of 38th IEE Symposium of Foundations of Computer Science*, 1997.

[17] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, and L. Shrira. Replication in the harp file system. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 226–238. ACM Press, 1991.

[18] K. M. Martin, R. Safavi-Naini, H. Wang, and P. R. Wild. Distributing the encryption and decryption of a block cipher. Cryptology ePrint Archive, Report 2003/005, 2003.

[19] M. Naor and B. Pinkas. Oblivious transfer with adaptive queries. In *Proceedings of Crypto'99*, 1999.

[20] M. Naor and B. Pinkas. Distributed oblivious transfer. *Lecture Notes in Computer Science*, 1976, 2000.

[21] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *Proceedings of SODA 2001 (SIAM Symposium on Discrete Algorithms)*, 2001.

[22] A. Serjantov. Anonymizing censorship resistant systems. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, 2002.

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

[24] S. W. Smith and D. Safford. Practical private information retrieval with secure coprocessors. In *Technical report, IBM T.J. Watson Research Center*, 2000.

[25] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of IEEE Symposium on Security and Privacy*, 2000.

[26] K. Yang. Private information retrieval for streaming data, 2001.