Usable PIR

Peter Williams and Radu Sion[*]
Network Security and Applied Cryptography Lab, Stony Brook University
E-mail: {petertw,sion}@cs.stonybrook.edu

Abstract

In [22] we showed that existing single-server computational private information retrieval (PIR) protocols for the purpose of preserving client access patterns leakage *are orders of magnitude slower than trivially transferring the entire data sets to the inquiring clients. We thus raised the issue of designing efficient PIR mechanisms in practical settings.*

In this paper we introduce exactly such a technique, guaranteeing access pattern privacy against a computationally bounded adversary, in outsourced data storage, with communication and computation overheads orders of magnitude better than existing approaches. In the presence of a small amount ($O(\sqrt{n})$, where n is the size of the database) of temporary storage, clients can achieve access pattern privacy with communication and computational complexities of less than $O(log^2 n)$ per query (as compared to e.g., $O(log^4 n)$ for existing approaches).

We achieve these novel results by applying new insights based on probabilistic analyses of data shuffling algorithms to Oblivious RAM [17], allowing us to significantly improve its asymptotic complexity. This results in a protocol crossing the boundary between theory and practice and becoming generally applicable for access pattern privacy. We show that on off the shelf hardware, large data sets can be queried obliviously orders of magnitude faster than in existing work.

1 Introduction

In an increasingly networked world, computing and storage services require security assurances against malicious attacks or faulty behavior. As networked storage architectures become prevalent – e.g., networked file systems and online relational databases in sensitive public and commercial infra-structures such as email and storage portals, libraries, health and financial networks – protecting the confidentiality and integrity of *stored* data is paramount to ensure safe computing. In networked storage, data is often geographically distributed, stored on potentially vulnerable remote servers or transferred across untrusted networks; this adds security vulnerabilities compared to direct-access storage.

Moreover, today, sensitive data is being stored on remote servers maintained by third party storage vendors. This is because the total cost of storage management is 5–10 times higher than the initial acquisition costs [12]. Moreover, most third party storage vendors do not provide strong assurances of data confidentiality and integrity. For example, personal emails and confidential files are being stored on third party servers such as Gmail [2], Yahoo Mail [4], Xdrive [3], and Files Anywhere [1]. Privacy guarantees of such services are at best declarative and often subject customers to unreasonable fine-print clauses—e.g., allowing the server operator (and thus malicious attackers gaining access to its systems) to use customer behavior for commercial profiling, or governmental surveillance purposes [9].

To protect data stored in such an untrusted server model, security systems should offer users assurances of data confidentiality and access pattern privacy. However, a large class of existing solutions delegate this by assuming the existence of co-operating, non-malicious servers. As a first line of defense, to ensure confidentiality, all data and associated meta-data can be encrypted at the client side using non-malleable encryption, before being stored on the server. The data remains encrypted throughout its lifetime on the server and is decrypted by the client upon retrieval.

Encryption provides important privacy guarantees at low cost. It however, is only a first step as significant amounts of information are still leaked through the access pattern of encrypted data. For example, consider an adversarial storage provider that is able to determine a particular region of the encrypted database corresponds to an alphabetically sorted keyword index. This is not unreasonable, especially if the adversary has any knowledge of the client-side soft-

[*]The authors are supported in part by the NSF through awards CT CNS-0627554, CT CNS-0716608 and CRI CNS-0708025. The authors also wish to thank Motorola Labs, IBM Research, the IBM Software Cryptography Group, CEWIT, and the Stony Brook Office of the Vice President for Research.

ware logic. The adversary can then correlate keywords to documents by observing which locations in the encrypted keyword index are updated when a new encrypted document is uploaded.

In existing work, one proposed approach for achieving access pattern privacy is embodied in Private Information Retrieval (PIR) [8]. PIR protocols aim to allow clients to retrieve information from public or private databases, without revealing to the database servers which record is retrieved. Recently however, we showed [22] that deployment of existing non-trivial single server PIR protocols on real hardware of the recent past would have been orders of magnitude more time-consuming than trivially transferring the entire database. Their deployment would in fact *increase* overall execution time, as well as the probability of *forward* leakage, when the present trapdoors become eventually vulnerable – e.g., today's queries will be revealed once factoring of today's values will become possible in the future. We stressed that this result goes beyond existing knowledge of mere "impracticality" under unfavorable assumptions. On real hardware, *no* existing non-trivial single server PIR protocol could have possibly had out-performed the trivial client-to-server transfer of records in the past, and is likely not to do so in the future either. This negative result is due to the fact that on any known past general-purpose Von Neumann hardware, it is simply more expensive to PIR-process one bit of information than to transfer it over a network.

A related line of research tackles client-privacy of accesses to client-originated data on a server. Specifically, the server hosts information for a client, yet does not find out which items are accessed. Note that in this setup the client has full control and ownership over the data and other parties are able to access the same data through this client only. One prominent instance of such mechanisms is Oblivious RAM (ORAM) [17]. For simplicity, in the following we will use the term ORAM to refer to any such outsourced data technique.

The recent advent of tamper-resistant, general-purpose trustworthy hardware (such as the IBM 4764 Secure Co-Processor [5]) opens the door to effectively deploying ORAM primitives for PIR purposes (i.e., for arbitrary public or private data, not necessarily originated by the current client) by deploying such hardware as a trusted server-side client proxy. The SCPU offers complete tamper detection, as well as remote code attestation to prove to clients that a particular program is in fact running unmodified on such a SCPU. However, trusted hardware devices are not a panacea. Their practical limitations pose a set of significant challenges in achieving sound regulatory-compliance assurances. Specifically, heat dissipation concerns under tamper-resistance requirements limit the maximum allowable spatial gate-density. As a result, general-purpose secure coprocessors (SCPUs) are often significantly constrained in both computation ability and memory capacity, being up to one order of magnitude slower than host CPUs.

In this paper we first introduce an efficient ORAM protocol with significantly reduced communication and computation complexities ($O(log^2 n)$ vs. $O(log^4 n)$ for [17]) – suited for deployment on constrained hardware such as SCPUs. We propose its deployment on existing secure hardware (IBM 4764 [5]) and show that the achievable throughputs are practical and orders of magnitude higher than existing work. These results constitute a first step to making PIR assurance truly practical.

2 Model

Deployment. In our discourse, we will consider the following concise yet representative interaction model. Sensitive data is placed by a client on a data server. Later, the client or a third party will access the outsourced data through an online query interface exposed by the server. Network layer confidentiality is assured by mechanisms such as SSL/IPSec. Without sacrificing generality, we will assume that the data is composed of equal-sized blocks (e.g., disk blocks, or database rows).

Clients need to read and write the stored data blocks while revealing a minimal amount of information (preferably none) to the server. We will describe the protocols here from the perspective of the client who will implement to primitives: read(id), and write(id, $newvalue$). Specifically, the (untrusted) server need not be aware of the protocol, but rather just provide traditional store/retrieve primitives (e.g., a file server interface).

Timing attacks have the potential to leak some information during the operation of this algorithm, if the algorithm is not implemented properly to avoid these attacks. We assume that the implementation is built in a manner that does not leak any secret information (such as the location of the fake blocks) through timing, noting that (1) any implementation can be turned into a timing-attack-free implementation simply by waiting longer on paths determined by secret information so that the length of all paths matches the length of the longest path, and (2) the transformation in (1) can be achieved without affecting the running time complexity of this algorithm.

Adversary. The adversarial setting considered in this paper assumes a server that is *curious but not malicious*. While it desires to illicitly gain information about the stored data, it nevertheless executes all queries in a correct manner. We are not concerned here with denial of service behavior. We also assume the adversary can be represented by a polynomial-time Turing machine; i.e., it is computationally bounded, thereby allowing us to take advantage of the following cryptographic primitives.

Cryptography. We require three cryptographic primitives with all the associated semantic security [16] properties: (i) a secure hash function which builds a distribution from its input that is indistinguishable from a uniform random distribution, (ii) an encryption function that generates unique ciphertexts over multiple encryptions of the same item, such that a computationally bounded adversary has no non-negligible advantage at determining whether a pair of encrypted items of the same length represent the same or unique items, and (iii) a pseudo random number generator whose output is indistinguishable from a uniform random distribution over the output space.

3 Related Work

PIR. Private Information Retrieval has been proposed as a primitive for accessing outsourced data over a network, while preventing its storer from learning anything about client access patterns [8]. In initial results, Chor et al.[8] proved that in an information theoretic setting in which queries do not reveal any information at all about the accessed data items, any solution requires $\Omega(n)$ bits of communication. To avoid this overhead, if multiple non-communicating databases can hold replicated copies of the data, PIR schemes with only sub-linear communication overheads are shown to exist [8]. For example, Sassaman et al.[21] applied such a scheme to protect the anonymity of email recipients. In the world of data outsourcing, in which there are only a few major storage providers, we do not believe the assumption of non-collusion among such untrusted servers is always practical, so we do not wish to rely on this assumption. Goldberg et al.[15] introduced a construction that combines a multi-server PIR scheme with a single-server PIR scheme, to guarantee information-theoretic PIR if the servers are not colluding, but still maintain computational privacy guarantees when all servers are colluding.

It is not our intention to survey the inner workings (beyond complexity considerations) of various PIR mechanisms or of associated but unrelated research. We invite the reader to explore a multitude of existing sources, including the excellent, almost complete survey by William Gasarch [13, 14]. Additionally we invite the reader to explore the results in [22] where we show that existing PIR protocols are orders of magnitude slower that trivially transferring the entire data sets to the inquiring clients.

It is worth noting that Asonov was the first to introduce [7] a PIR scheme that uses a secure CPU to provide (an apparent) $O(1)$ online communication cost between the client and server. However, this requires the secure CPU on the server side to scan the entire database on every request, indicating a hidden computational complexity cost of $O(n)$, where n is the size of the database.

ORAM. Oblivious RAM [17] provides access pattern privacy on a database, requiring only logarithmic storage. The amortized communication and computational complexities are both $O(log^3 n)$, or $O(log^4 n)$ in practice because the asymptotic notation hides a very large constant factor in the $O(log^3 n)$ implementation. A variation of ORAM is implemented by Iliev and Smith [18], who deploy secure hardware to obtain PIR at a cost of $O(\sqrt{n}\log n)$. This is better than the poly-logarithmic complexity granted by Oblivious RAM for the small database sizes they consider. This work is notable as one of the first full ORAM-based PIR setups, albeit with lower query throughputs.

A PIR mechanism with $O(n/k)$ costs is introduced by Wang et al.in [23], where n is the database size and k is the amount of secure storage. The protocol is based on a careful scrambling of a minimal set of server-hosted items. A partial reshuffle costing $O(n)$ is performed every time the secure storage fills up, which occurs once every k queries. While a significant improvement, this result is not always practical since the total database size n often remains much larger than the secure hardware size k. In practice, hard disk capacity (and enterprise database size) is increasing faster than secured memory capacity, which is severely limited by space and heat dissipation constraints inside a secure CPU.

In this paper we introduce a solution with only $O(\log^2 n)$ amortized overhead, in the presence of $c\sqrt{n}$ temporary client storage, where c is an independent security parameter. We show this to be a (first) solution that can be implemented efficiently over large data sets.

3.1 Oblivious RAM Overview.

Since the proposed protocol is based on ORAM [17], a brief summary of the operation of ORAM follows. The database is considered a set of n encrypted blocks and supported operations are read(id), and write(id, $newvalue$). The data is organized into $log_4(n)$ levels, pyramid-like. Level i consists of up to 4^i blocks. Each block is assigned to one of the 4^i buckets at this level as determined by a hash function. Due to hash collisions each bucket may contain from 0 to $\log n$ blocks.

Reading. To obtain the value of block id, the client must perform a read query in a manner that maintains two invariants: (i) it must never reveal which level the desired block is at, and (ii) it must never look twice in the same spot for the same block. To maintain (i), the client always scans a single bucket in every level, starting at the top (Level 0, 1 bucket) and working down. The hash function informs the client of the candidate bucket at each level, which the client then scans. *Once the client has found the desired block, the client still proceeds to each lower level, scanning random buckets instead of those indicated by their hash function*. For (ii), once all levels have been queried, the client then re-encrypts the query result with a different nonce and places it in the *top* level. This ensures that when the client repeats a search for this block, it will locate the block immediately (in a different location), and the rest of the search pattern will be randomized. Note that the top level will quickly fill up; the process to dump the top level into the one below is described later.

Writing. Writes are performed identically to reads in terms of the data traversal pattern, with the exception that the new value is inserted into the top level at the end. Inserts are performed identically to writes, since no old value will be discovered in the query phase. Note that semantic security properties of the re-encryption function ensure the server is unable to distinguish between reads, writes, and inserts, since the access patterns are indistinguishable between each case.

Level Overflowing. Once a top level is full, it is emptied into the level below, and this second level is completely re-encrypted, and re-ordered according to a new hash function. Thus, accesses to this new iteration of the second level will hence-forth be completely independent of any previous accesses. Note that each level will overflow once the level above it has been emptied 4 times. Any re-ordering must be performed obliviously: once complete, the adversary must be unable to make any correlation between the old block locations and the new locations. A sorting network is used to reorder the blocks.

To enforce invariant (i), note also that all buckets must contain the same number of blocks. For example, if the bucket scanned at a particular level has no blocks in it, then the adversary would be able to determine that the desired block was *not* at that level. Therefore, each reorder process fills all partially empty buckets up to the top with *fake* blocks. Recall that since every block is encrypted with semantic security, the adversary cannot distinguish between fake and real blocks.

Costs. Each query requires a total online cost of $O(\log^2(n))$ for scanning the $\log n$-sized bucket on each of the $\log n$ levels, plus an additional, amortized cost due to intermittent level overflows. Using a logarithmic amount of client storage, reshuffling levels in ORAM requires an amortized cost of $O(\log^3(n))$ per query ($O(\log^4(n))$ in practice due to a hidden constant factor around 2^{100} in the implementation [17]).

4 A Solution

Our solution deploys new insights based on probabilistic analyses of data shuffling in ORAM allowing a significant improvement of its asymptotic complexity. These results can be applied under the assumption that clients can afford a small $O(\sqrt{n})$ amount of temporary working memory. We validate this assumption in real settings (e.g., the IBM 4764 SCPU [5] can host up to 64MB of RAM).

4.1 Additional Client-side Working Memory

Simply adding storage to ORAM in a straightforward manner does not improve its complexity. Consider that there are two stages where additional storage can be deployed. First, the top levels could be stored exclusively on the client, allowing the bypassing of all reads and writes to the top levels, as well as the re-shuffling of these levels. Since there are $\log n$ levels, with sizes 4^i for i from 1 to $\log n$, the blocks belonging to the first $\log(c\sqrt{n}) = \log c + \frac{1}{2}\log n$ levels can fit in this storage. This however, would only eliminate a constant fraction of the levels, leaving the most expensive levels operating as before.

Second, as indicated in [17], additional client-side storage can be deployed in the sorting network used in the level reshuffle. The sorting network is the primitive that performs all the level reordering, requiring $O(n \log^2 n)$ time for the client to obliviously sort data on the server (with no client storage). The ORAM claim of $O(\log^3 n)$ amortized overhead requires the use of the impractical AKS sorting network [6] that performs in $O(n \log n)$ time (with a hidden factor of close to 2^{100}).

Then, in the presence of additional storage, the normal sorting network running time can be improved by performing comparisons in batches on the client. However, performing batch comparisons with a limited amount of storage does not greatly improve the complexity of the sorting network. We are not aware of methods to apply this amount of storage that would result in improvements of more than a constant factor. While we cannot make a claim of nonexistence of such improvements, we can bound the degree of improvement possible by this approach. Even if the storage is used in a manner that can reduce the time complexity of the sorting sequence, no amount of storage can cause the sorting network to do a comparison sort (as required in Oblivious RAM) better than $\Omega(n \log n)$ [10]. This still results in overall amortized overhead of $\Omega(\log^3 n)$.

Our Approach. Instead, we propose to tackle the complexity of the most time-consuming phase of ORAM, the level reorder step. We take advantage of the consistent nature of uniform random permutations to perform an oblivious scramble with a low complexity and little client storage. Our intuition is that given two halves of an array consisting of uniformly randomly permuted sequence of items, the items will be distributed between the halves almost evenly. That is, if we pick the permuted items in order, counting the number of times each array half is accessed, the counts for each array half remain close for the entire sequence, with high probability.

This allows us to implement a novel merge sort that hides the order in which items are being pulled from each half. Once the two array halves are each sorted and stored on the server, we can combine them into a sorted whole by reading from each half into the client buffer, then outputting them in sorted order without revealing anything about the permutation. By the uniform nature of the random permutation, for arrays of size n, we show that the running tally of picks from each array half will never differ by more than $c\sqrt{n}$, with high probability. This means that we can pre-set a read pattern from the server without knowing the permutation, and still successfully perform the permutation! The pattern of accesses between the two array halves will deviate slightly, but with high probability they will fall within the window of $c\sqrt{n}$ from the fixed pattern.

This oblivious merge sort is the key primitive that allows us to implement access pattern privacy with $O(\log^2(n))$ overhead. We use it to implement a random scramble, as well as to remove the fake blocks that are stored in each level. Being able to do both of those steps efficiently, we can then replace the oblivious permutation used in ORAM with a more efficient version.

From here on, we will be concerned mainly with the process of re-ordering a level, since the rest of our algorithm is unchanged from ORAM. A level re-ordering entails taking the entire contents of level i, consisting of 4^i buckets sized $\log n$, containing a total number of real blocks between 4^{i-1} and 4^i, with the remainder filled with fake blocks, and rearranging them to the new permutation obliviously – without revealing anything about the new permutation to the server storing these levels.

4.2 Strawman: client with n blocks of working memory

Before describing the main result, let us first analyze a strawman algorithm that achieves our desired time complexity, in the presence of enough client-sided storage to fit the *entire* database.

Observe that if the client has $n \geq 4^i$ blocks of temporary secure storage, it can perform a level reorder with $(2)(4^i)(\log n) = O(\log n 4^i)$ server accesses. By reading the entire level into the temporary secure storage, throwing out the fake blocks as they were encountered, it can store all 4^i blocks locally. It then performs a comparison sort on the local secure storage (which is hence done without revealing the new permutation to the server) to permute these blocks to their new location at a computational cost of $O(4^i \log(4^i)) = O(i4^i)$. The blocks are then all re-encrypted with new nonces for a cost of $O(4^i)$ (so the server is unable to link old to new blocks). Copying this data back to the server, while inserting fakes to fill the rest of the buckets, requires writing another $(4^i)(\log n)$ blocks to the server.

Since each level i overflows into level $i+1$ once every 4^i accesses, level $i+1$ must be reordered at each such occurrence. As there are $\log n$ levels total, the amortized communication and computational costs per query of this level reordering approach, across all levels, can therefore be approximated by

$$\sum_{i \leftarrow 1}^{\log n} \frac{O(\log n 4^i)}{4^{i-1}} = \sum_{i \leftarrow 1}^{\log n} O(\log n) = O(\log^2 n)$$

This offline level reordering cost must be paid in addition to the online query cost to scan a bucket at each level. This part of the algorithm is equivalent to ORAM. Since the buckets have size $\log n$, the online cost of scanning buckets is $(\log_4 n)(\log n) = O(\log^2 n)$. Thus the average cost per query, including both online costs and amortized offline costs, is $O(\log^2 n)$.

In summary, in the presence of $O(n)$ client storage the amortized running time for ORAM can be cut down from $O(\log^4 n)$ to $O(\log^2 n)$. Of course, assuming that the client has n blocks of local working memory is not necessarily practical and could even invalidate the entire cost proposition of server-hosted data. Thus little has been gained so far.

4.3 Overview: client with only $c\sqrt{n}$ blocks of working memory

We will now describe an algorithm for level re-ordering with identical time complexity, but requiring only $c\sqrt{n}$ local working memory from the client. The client's reordering of level i is divided into Phases (refer to Figure 1). We now overview these phases and then discuss details.

1. **Removing Fakes.** Copy the 4^i original data blocks at level i to a new remote buffer (on the server), obliviously removing the $(\log n - 1)4^i$ fake blocks that are interposed. Care must be taken to prevent revealing which blocks are the fakes – thus copying will also entail their re-encryption. This decreases the size of the working set from $(\log n)4^i$ to 4^i if the level is full, or to $\frac{1}{4}4^i$, $\frac{1}{2}4^i$, or $\frac{3}{4}4^i$ for the first, second, and third reorderings of this iteration of level i. We will assume we are dealing with a full level (fourth reordering) to make the remainder of this description simpler; earlier reorderings proceed equivalently but with slightly lesser time and space requirements. The communication/computational complexity of this phase is $O(\log n 4^i)$. (see section 4.4)

2. **Oblivious Merge Sort.** Obliviously merge sort the working set in the remote buffer, placing blocks into their final permutation according to the new hash function for this level. Perform the merge sort in such a way that the server can build no correlation between the original arrangement of blocks and the new permutation. The communication/computational complexity of this phase is $O(\log n 4^i)$. (see section 4.5)

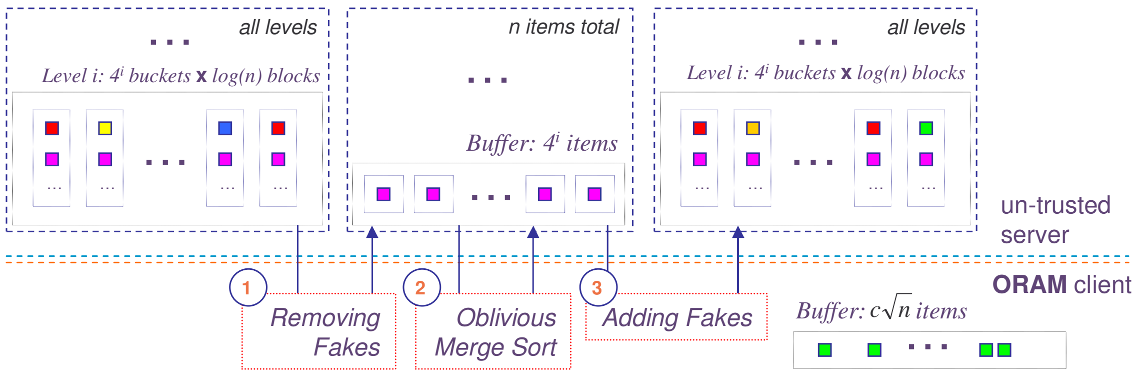

Figure 1. Solution Overview.

3. **Add Fakes.** Copy the 4^i blocks, which were permuted by Phase 2 into their correct order, to the final remote storage area for level i. They are not in buckets yet, so we build buckets, obliviously adding in the $(\log n - 1)4^i$ fake blocks necessary to guarantee all buckets have the same size. The communication/computational complexity of this phase is $O(\log n4^i)$. (see section 4.6)

The above algorithm reorders level i into the new permutation, in time $O(\log n4^i)$. Therefore the derivation of the amortized overhead is equivalent to the derivation performed for the strawman algorithm, leading to an amortized overhead of $O(\log^2 n)$ per query. We now show how to efficiently implement each phase, using only $c\sqrt{n}$ local memory.

4.4 Phase 1: Remove Fakes

Fake blocks can be removed from level i in a single pass, without revealing them, by copying into a temporary buffer that hides the correspondence between read blocks and output blocks (refer to Figure 2 (a)). The client scans the level, storing the real blocks into a local queue and tossing the fake blocks. Once the queue is expected to be *half full*, the client starts writing blocks from the queue (while also continuing the scan), at a rate corresponding to the overall ratio of real to fake blocks. The goal is to keep the queue about half full until the end. (The server can observe the total number of fake and real blocks in a particular level, which is independent of the data access pattern.) Assuming the temporary queue never overflows or empties entirely until the end, the exact pattern of reads and writes observed by the server is dependent only on the number of blocks, and the ratio of fakes. The server learns nothing of which are the fake blocks by observing the fake removal scan. We show in Theorem 1 that, with high probability, a queue of size $c\sqrt{n}$ will not overflow or empty out.

```
$remove\_fakes\_from\_level()$
s ← maximum size of local queue, $c\sqrt{n}$
q ← empty queue stored locally, size $s$
r ← ratio of fake blocks to real blocks
for x = 1 to r*(n+s/2) do
    if $x < r * n$ then
        t ← decrypt(readNextBlockFromLevel());
        if t is a real block then
            enqueue(q, t)
        end if
    end if
    if total number of blocks output $< x/r - s/2$ then
        t ← dequeue(q)
        writeNextBlockToRemoteBuffer(encrypt(t))
    end if
end for
```

```
$add\_fakes\_to\_level()$
s ← size of local queue, $c\sqrt{n}$
q ← empty queue stored locally, size $s$
r ← ratio of buckets to real blocks, determined
by the reshuffle number for this level.
c ← 0 (total number of buckets output so far)
for x =1 to n + s/2 do
    if $x \leq n$ then
        t ← decrypt(readNextBlockFromLevel());
        enqueue(q, t)
    end if
    if $c < r * (x - s/2)$ then
        $c \leftarrow c + 1$
        items ← Dequeue all items corresponding to bucket c.
        (They will be on the end of the queue if there are any)
        b ← New bucket, containing those items, and filled
        the rest of the way with fake blocks.
        writeNextBucketToRemoteLevel(encrypt(b))
    end if
end for
```

Figure 2. (a) Phase 1: Remove Fakes (b) Phase 3: Add Fakes

```
$oblivious\_merge\_sort(A)$
if A is size 1 then
    Return A
end if
A1 ← First Half of A
A2 ← Second Half of A
A1 ← ObliviousMergeSort(A1)
A2 ← ObliviousMergeSort(A2)
B ← New remote buffer with the same size as A
s ← size of local queues, $c\sqrt{n}$
q1 ← empty queue stored locally, size $s$
q2 ← empty queue stored locally, size $s$
for x = 1 to s/2 do
    Enqueue(q1, decrypt(readNextBlockFrom(A1)));
    Enqueue(q2, decrypt(readNextBlockFrom(A2)));
end for
(At this point, each queue will have s/2 blocks)
for x = s/2 to n + s/2 do
    if $x \leq n$ then
        Enqueue(q1, decrypt(readNextBlockFrom(A1)));
        Enqueue(q2, decrypt(readNextBlockFrom(A2)));
    end if
    (Now we've read 2 blocks; time to output 2 blocks)
    for i = 1 to 2 do
        t1 ← Peek(q1);
        t2 ← Peek(q2);
        if HashLocation(t1) > HashLocation(t2) then
            t ← dequeue(q1)
        else
            t ← dequeue(q2)
        end if
        writeNextBlockTo(B, encryptWithNewNonce(t));
    end for
end for
Return B
```

Figure 3. Phase 2: Oblivious Merge Sort

This phase requires time linear to the size of the level being read. For level i, which contains 4^i buckets of size $\log n$, the running time is $O(4^i \log n)$.

Since the location of real blocks is determined by a secure hash function on the unique block index, the distribution of the blocks is indistinguishable from a uniform random distribution. Hence the fake blocks, as well, will be spread uniformly randomly across the entire level. With high probability, any sample of buckets will thus have a ratio of fake blocks to real blocks very close to the overall average. Before we formulate this, we introduce the following lemma:

Lemma 1. *A one-dimensional random walk consisting of n steps of size 1, either forward or backward with equal probability, will remain bounded by $\pm c\sqrt{n}$ with high probability.*

Proof. (sketch) Let the likelihood of a such a one-dimensional random 50-50 walk being at location d at step j be defined as $P_j(d)$. Then the likelihood that the walk is at position $c\sqrt{n}$ or $-c\sqrt{n}$ at step j is $2P_j(c\sqrt{n})$ and the likelihood that any step j along the way hits either of these out-of-bounds markers becomes: $2\sum_{j\leftarrow 1}^{n} P_j(c\sqrt{n})$. [11] uses Stirling's formula to approximate $P_j(d) \approx \frac{2}{\sqrt{2\pi j}}e^{-d^2/2j}$ For $1 \leq j < n$, $P_j(c\sqrt{n}) < P_n(c\sqrt{n})$. Therefore,

$$
\begin{aligned}
2\sum_{j\leftarrow 1}^{n} P_j(c\sqrt{n}) < 2nP_n(c\sqrt{n}) &\approx 2n\frac{2}{\sqrt{2\pi n}}e^{-(c\sqrt{n})^2/2n} \\
&= 4\sqrt{\frac{n}{2\pi}}e^{-c^2/2}
\end{aligned}
$$

For any fixed maximum walk length n_{max}, the chance of reaching $\pm c\sqrt{n}$ in a 50-50 random walk is negligible with the security parameter c. $\square$

Theorem 1. *With high probability, the Remove Fakes queue never overflows or empties early.*

Proof. (sketch): Let $r = O(\log n)$ be the ratio of fake blocks to real blocks in the bottom level. Consider a random walk of length nr. With probability $1/r$, we step forward $(1 - 1/r)$. With probability $1 - 1/r$, we step backwards $1/r$. Then the main idea behind this proof is to reduce the problem to showing that such a random walk will remain within $c\sqrt{n}$ of the starting location with high probability. It can then be shown that this in turn can be reduced to the probability of a standard 50-50 +1/-1 walk length n leaving these bounds. $\square$

To summarize, Phase 1 copies all of the real blocks out of level i, into a new remote (server-side) storage buffer that only contains real blocks. In copying, a small local (client-side) buffer is used to avoid leaking which blocks were fake. This is possible since the fake blocks are evenly distributed throughout the level.

4.5 Phase 2: Oblivious Merge Sort

We now describe an algorithm that performs a merge sort on a array of size n, with $c\sqrt{n}$ local storage, in $O(n \log n)$ time, without revealing any correlation between the old and new permutations. The algorithm runs recursively on the remote array as described in Figure 3. The recursion depth is $\log n$, and each level of recursion entails a single pass of size $O(n)$ across the entire array.

The correctness of this algorithm depends on the uniformity of the starting permutation of the items being sorted, as illustrated in Theorem 3. Its oblivious nature derives immediately by construction:

Theorem 2. *The Oblivious Sort algorithm is private: no more than a negligible amount of information about the new permutation is leaked to a computationally bounded adversary.*

Proof. (sketch): The ordering of reads and writes in every instantiation of the scramble is identical: observe that in the algorithm defined in Figure 3, the readNextBlockFrom() and writeNextBlockTo() functions are called in the same pattern every time, depending only on n, not the comparisons made on the HashLocation()s.

The semantic security properties of the symmetric encryption scheme guarantee that the adversary cannot correlate any two blocks based on the encrypted content (the server cannot determine whether t is from $q1$ or $q2$). Therefore, every instantiation of the scramble appears identical to the server: it sees a fixed pattern of reads interspersed with a fixed pattern of writes of unintelligible data. The specific fixed pattern is known beforehand to the server (from the algorithm definition), and the content of the reads has no correlation to the content of the writes since the blocks are re-encrypted with a semantically secure encryption scheme at the client.

Therefore, in observing any iteration (or sequence of iterations) of the oblivious merge sort, the (computationally bounded) adversary learns nothing.

Moreover, the final permutation is chosen from among all possible permutations. Since the access pattern is identical when generating each of these permutations, the server has no ability to guess the resulting permutation.

A small number of permutations will cause the algorithm to fail and output $\perp$, if the queues overflow, but this absence of a failure reveals only a negligible amount of information about the new permutation, since failure occurs with very low probability, as shown next. □

Theorem 3. *Oblivious Merge Sort queues never overflow or empty early, with high probability.*

Proof. (sketch): The queue size at step j is a probabilistic function Q_j defined iteratively:

$$\begin{cases} Q_0 = n/2 \\ Q_j = Q_{j-1} + 1 \quad \text{Pr.} \quad 1/2 - (Q_{j-1} - n/2)/(n-j) \\ Q_j = Q_{j-1} - 1 \quad \text{Pr.} \quad 1/2 + (Q_{j-1} - n/2)/(n-j) \end{cases}$$

This is analogous to pulling two colors of balls out of a bag without replacement, starting with $n/2$ of each color in the bag. The further we deviate from an equivalent number of each color, the more likely it is for the next ball to bring the tally closer to equivalent counts. This negative dependency implies that any step away from the balance will occur with probability asymptotically lower than $1/2$. This can be then reduced to showing that a 50-50 random walk will remain within $\pm c\sqrt{n}$ with high probability and Lemma 1 etc. □

In summary, the Oblivious Sort algorithm sorts all the data blocks on the server into their final permutation, without revealing anything that could allow the server to correlate the two permutations.

4.6 Phase 3: Add Fakes

In the final phase, the permuted blocks are added back to server-hosted buckets where they will be located by the next iteration of the secure hash function (see Section 3.1). At the same time, fake blocks are added to make all buckets mutually indistinguishable. This is the exact inverse of Phase 1.

For correctness we must also show here that the buckets of size $\log n$ will not overflow. A simple balls and bins result shows that if 4^i balls are randomly thrown into 4^i bins, with probability greater than $\frac{n-1}{n}$, the fullest bin has fewer than $3\frac{\log 4^i}{\log\log 4^i}$ balls when n is large enough [20]. This is small but non-negligible probability. If this case ever occurs, the authors of [17] prescribe a level re-order abort and restart. With high probability, the time complexity of this algorithm is not affected. See [17] for a more complete analysis of this issue. We note that while this restriction reveals to the server that the hash function finally chosen does not overflow any bucket, it does not reveal any correlation between previous and current block locations.

As in Phase 1, we employ a local buffer of size $c\sqrt{n}$ to prevent the server from learning where fakes are being added. The client scans the array of real blocks stored in the remote server by Phase 2 into a local queue. Once the local queue is half full, it begins constructing server-side buckets with the blocks from the queue, writing into one bucket for every block read. As long as the temporary queue does not overflow or become empty, the exact pattern of reads and writes observed by the server is dependent only on the number of blocks. Therefore, the server learns nothing of which are the fake blocks by observing this process (see Figure 2 (b)). Moreover, it does not reveal the number of blocks in each bucket, since the buckets are written sequentially to the server in full, so the read and write pattern for this step is identical on every repetition.

The algorithm runs in time linear to the size of the level being written. For level i, which contains 4^i buckets of size $\log n$, the running time is $O(\log n 4^i)$.

Theorem 4. *With high probability, the Add Fakes algorithm queue never overflows or empties early.*

Proof. (sketch): The queue length Q_j at step j is modeled by $Q_{j+1} = Q_j + 1 - bucketSize(j)$. The bucket sizes, determined by a fair balls and bins experiment placing n balls into n bins, are distributed according to a Poisson distribution parameter 1 [20]. Taking the sum of j Poisson random variables yields a Poisson random variable [20]. The walk distance after j steps can therefore be modeled as j minus a Poisson random variable X with mean j etc. □

Theorem 5. ***Correctness.*** *After Phase 3, all blocks will be in the correct bucket (determined by the secure hash function).*

Proof. (sketch): This proof follows from the construction of Phases 2 and 3. Phase 3 correctly builds the buckets for level i when its input array satisfies the follow properties: (1) all data blocks corresponding to i are in the array. (2) For all data blocks b, b', if the bucket corresponding to data block b precedes the bucket corresponding to data block b', then b is listed in the array before b'. After the sort in Phase 2, all blocks are in sorted order, according to their bucket, therefore meeting the two requirements for the input to Phase 3. □

Theorem 6. ***Privacy.*** *The contents of the level make it from the old permutation to the new permutation without revealing any non-negligible information about either permutation. The location of the fake blocks is not revealed.*

Proof. (sketch): Theorem 2 shows that the level permutation performed in Phase 2 does not reveal any correlation between the old locations and the new locations. Furthermore, the read and write pattern of Phase 3 is independent of the data items and the final permutation, so Phase 3 does not reveal anything about the location of the fake blocks, or the permutation. □

5 Performance

	Server	Client	IBM 4764
RAM	4GB	1GB	32MB
processor		2Ghz	266Mhz
disk seek time	5ms		
sustained disk read/write	50 MB/s		
Link bandwidth		10 MB/s	80MB/s[a]
Link round trip time		50ms	0.1ms
En/Decryption		100MB/s[b]	10MB/s
Outsourced data set size	1 TB, in 1000-byte blocks; $n = 10^9$		

[a]The 4764 sits on an 8GB/s PCI-X bus; the bottleneck is the DMA rate.
[b]Based on processor speed, using AES [19].

Figure 4. Configuration used to compute sample values in the following tables and graphs.

In evaluating the feasibility and performance of the architecture we consider the sample configuration illustrated in Figure 4. Further, Figure 7 illustrates multiple such data points.

Online Cost. The query requires online scans of one bucket at each level, plus a write to the top level. The scan of the $\log_4 n$ levels are interactive; the bucket scanned at each level depends on the results of the previous level. Figure 5 displays the expected online cost per query.

It is clear from these estimates that in a sequential access model, the network latency is responsible for most of the query latency. This is due to the interactive nature of the scans; the client cannot determine the next bucket to scan until it has seen the contents of the previous.

	Formula	Sample
Network latency	$RTT_{link} * \log_4 n$	750ms
Disk seek	$Latency_{seek} * \log_4 n * 2$	150ms
Network transfer	$\log_4 n * \log n * 2 * blksz/Throughput_{link}$	60ms
Client en/decryption	$\log_4 n * \log n * 2 * blksz/Throughput_{crypto}$	6ms
Server disk transfer time	$\log_4 n * \log n * 2 * blksz/Throughput_{disk}$	12ms

Figure 5. Online cost per query, resulting from scanning a bucket at each level.

Offline Reorder Cost. The offline cost resulting from the reordering of level i (performed once every 4^{i-1} accesses) consists of three phases including a sequential level scan of size $\log n * 4^i$, and a sequential write-back of size 4^i to remove fakes (Phase 1). The oblivious sort (Phase 2) consists of $\log 4^i$ sequential scans of size 4^i. Adding fakes (Phase 3) requires requires copying back $\log n * 4^i$ items. To estimate this cost we must sum over all $\log_4 n$ levels, recalling that each level is reordered only once every 4^{i-1} queries. We therefore remove a factor of 4^{i-1}, and sum over all levels, to calculate the amortized overhead. Figure 6 shows the resulting formulas. If all of these costs are incurred sequentially, we have an amortized response rate of approximately $637ms/query$ offline plus $978ms/query$ online, for $1.6s/query$.

	Formula	Sample
Network latency. [a]	n/a	$<$ 1ms
Network transfer [b]	$\log_4 n * \log_n *4 * 4 * blksz/Throughput_{link}$	500ms
Disk seek latency [c]	n/a	$<$ 1ms
Disk transfer [d]	$\log_4 n * \log_n *4 * 4 * blksz/Throughput_{disk}$	98ms
Client processing	$\log_4 n * \log_n *4 * 4 * blksz/Throughput_{crypto}$	49ms

[a]Level reordering is not interactive, so idling can be avoided here.
[b]The Phase 1 and 3 scans account for its bulk.
[c]Seek time will be hidden by disk transfer during reordering.
[d]This load can be split among several disks.

Figure 6. Amortized **offline** cost per query.

The bottleneck when determining the parallel query throughput is the network throughput, at $560ms/query$. This results in a query throughput of just under 2 queries per second.

By comparison, in ORAM, the network transfer time alone for reshuffling level i consists of about 10 sorts of $4^i \log n$ data, each sort therefore requiring $4^i \log(n) \log^2(4^i \log n)$ block transfers, for a total of $10 * 4^i * \log(n) * log^2(4^i \log n) * 2^{10}/10MB/sec$. Summing over the $\log_4 n$ levels, and amortizing each level over 4^{i-1} queries, ORAM has an amortized network traffic cost per query of $\sum_{i \leftarrow 1}^{15} 10 * 4 * 15 * \log^2(15 * 4^i) * 2^{10}B = 614KB \sum_{i \leftarrow 1}^{15} (\log 15 + \log 4^i)^2 \approx 3.680GB$. Over the considered 10MByte/s link this results in a 368 sec/query amortized transfer time, almost three orders of magnitude slower.

Achieving PIR. So far we have described how to implement an ORAM-type of mechanism providing access pattern privacy for private data. A general PIR implementation

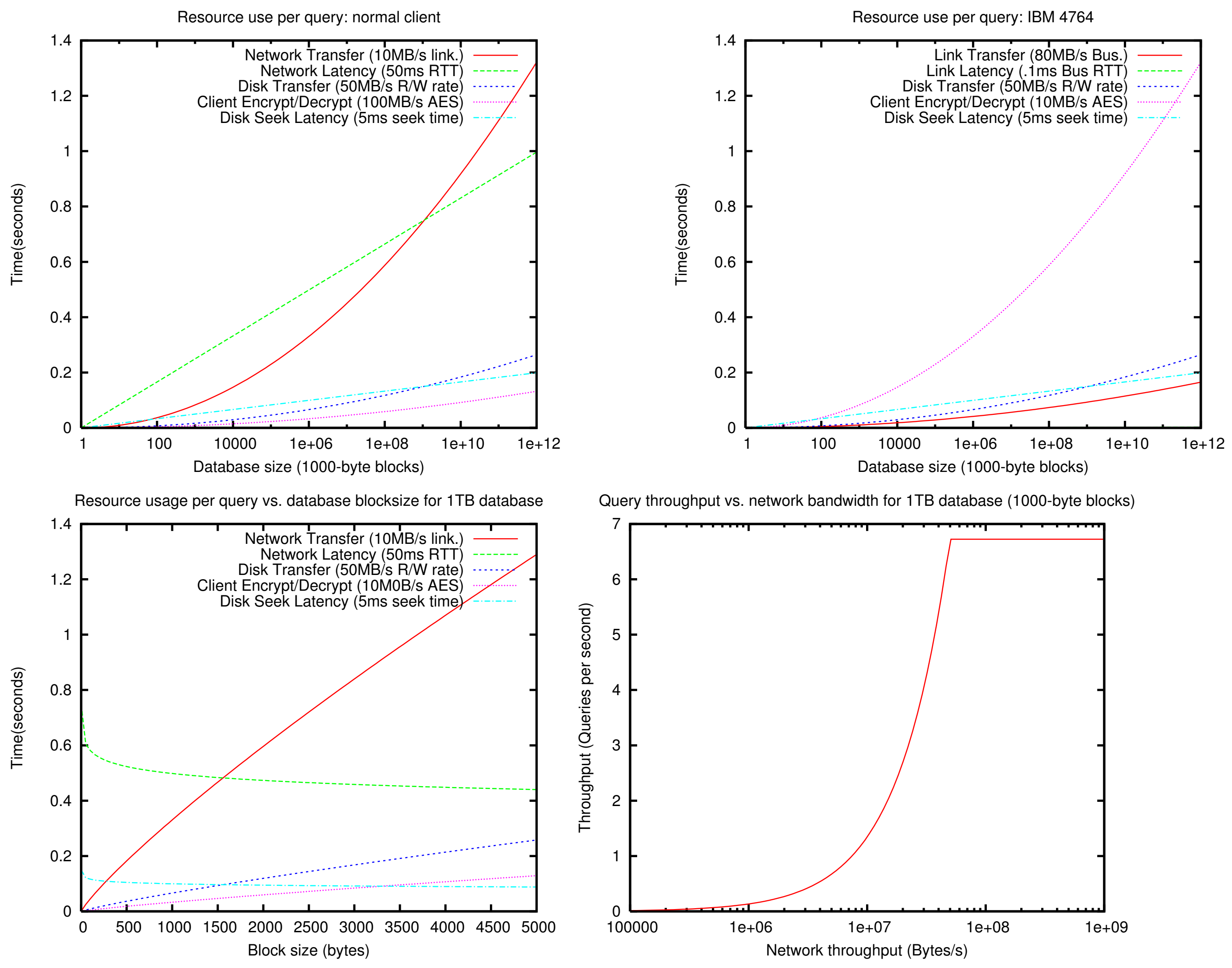

Figure 7. Amortized use of resources per query (a) for a normal client and (b) using the IBM 4764 SCPU as a client. (c) resource usage as the the database block size is varied, and (d) estimated query throughput as the the network link capacity is varied. Both (c) and (d) assume a fixed database size of 1TB.

requires a client to be able to download from a public server, meaning the client does not have access to prearranged secret keys. As discussed in Section 1 by implementing the access pattern privacy on a SCPU, we can achieve general PIR. The secure CPU maintains the encrypted database, and never leaks any of the encryption keys. Clients who wish to retrieve an item privately then interact with the main data through the SCPU.

Figure 7 shows that when we implement PIR on the secure CPU, the bottleneck is no longer the network bandwidth, but the en/decryption times. Under our sample configuration, the new bottleneck matches the old one, leaving us at 560ms per query, or just under 2 queries per second. Additionally, of concern is the limited SCPU storage. By setting c (the security parameter) to 10, corresponding to a chance of failure less than $4\sqrt{\frac{n}{2\pi}}e^{-c^2/2} < 4\sqrt{\frac{2^{50}}{2\pi}}e^{-10^2/2} < e^{-25}$, 32MB of RAM available on a SCPU can support databases of sizes up to 10TB, for 1 byte blocks, or 10GB, for 1 KB blocks.

Memory Pooling. A key advantage to our algorithm is that the working buffers are only used for a small period of time and are transient, thus requiring no backup. Therefore the high cost of client storage maintenance is avoided since no data is lost if the working memory is lost. A second advantage is that resources can be pooled between SCPUs to support larger databases. For example, if a storage provider manages 10 SCPUs for 10 customers, and if the working buffer is only in use for 10% of the time, the provider can pool the secure storage between SCPUs, allowing for an effective secure storage area of 320 MB instead of just 32 MB. This would allow the provider to support databases of size 10^{15} 1-byte blocks per client, or 1 TB if consisting of 1-KB blocks.

The limiting factor in pooling is the percentage of time the secure CPUs are put to use. This will vary based on the actual transaction patterns of the clients. If transactions are run continuously at the maximum throughput, we expect the idle time to be around 50%. If there are idle periods, however, and the average throughput is lower, each SCPU may see a much higher idle time. Note that if a provider overestimates the idle time of the CPUs, by supporting larger customer databases, the performance will suffer as clients must wait for each others' SCPUs to become free.

Existing PIR. Trivial PIR (transfering the entire database to the SCPU for every query) will have a bottleneck shared by the bus transfer time and the disk transfer time, of 50MB/sec. For our 1TB database, this will require about 22000 seconds per query [1].

The PIR protocol introduced in [23] offers an amortized complexity of $O(n/k)$ for database size n and secure storage size k. For $k = O(\sqrt{n})$, this yields an overhead of $O(\sqrt{n})$ per query. This is proving to be a reasonable estimate of k, since as described earlier in this paper, database sizes and hard disk capacity are increasing much faster than secured storage capacity. As databases become larger, our superior $O(log^2(n))$ complexity becomes increasingly necessary for obtaining practicality.

6 Conclusions

We introduced a (first) practical PIR mechanism, orders of magnitude faster than existing mechanisms. We have analyzed its overheads and security properties. We validated its practicality by exploring achievable throughputs on current off the shelf hardware. In future work we believe it is important to increase achievable throughputs. We are looking for ways to de-amortize the offline level reorder cost. Moreover, as the bulk of the overhead in this technique is related to the fake blocks, we are currently exploring alternate constructions that hide which level is accessed for a particular query, potentially bringing the amortized overhead to $O(\lg n \lg\lg n)$ per query.

7 Acknowledgments

We would like to thank Bogdan Carbunar for numerous discussions and feedback, as well as our anonymous reviewers, who offered helpful insights.

[1]We have already shown that no existing non-trivial PIR mechanism can be faster than the trivial PIR case [22].

References

[1] FilesAnywhere. Online at http://www.filesanywhere.com/.

[2] GMail. Online at http://gmail.google.com/.

[3] Xdrive. Online at http://www.xdrive.com/.

[4] Yahoo Mail. Online at http://mail.yahoo.com/.

[5] IBM 4764 PCI-X Cryptographic Coprocessor (PCIXCC). Online at http://www-03.ibm.com/security/cryptocards/pcixcc/overview.shtml, 2006.

[6] M. Ajtai, J. Komlos, and E. Szemeredi. An o(n log n) sorting network. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 1–9, 1983.

[7] D. Asonov. *Querying Databases Privately: A New Approach to Private Information Retrieval*. Springer Verlag, 2004.

[8] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *IEEE Symposium on Foundations of Computer Science*, pages 41–50, 1995.

[9] CNN. Feds seek Google records in porn probe. Online at http://www.cnn.com, January 2006.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill, 2001.

[11] Michael Fowler. A one-dimensional random walk. Online at http://galileo.phys.virginia.edu/classes/152.mf1i.spring02/RandomWalk.htm.

[12] Gartner, Inc. Server Storage and RAID Worldwide. Technical report, Gartner Group/Dataquest, 1999. www.gartner.com.

[13] W. Gasarch. A WebPage on Private Information Retrieval. Online at http://www.cs.umd.edu/~gasarch/pir/pir.html.

[14] W. Gasarch. A survey on private information retrieval, 2004.

[15] Ian Goldberg. Improving the Robustness of Private Information Retrieval. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, May 2007.

[16] O. Goldreich. *Foundations of Cryptography*. Cambridge University Press, 2001.

[17] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious ram. *Journal of the ACM*, 45:431–473, May 1996.

[18] A. Iliev and S.W. Smith. Private information storage with logarithmic-space secure hardware. In *Proceedings of i-NetSec 04: 3rd Working Conference on Privacy and Anonymity in Networked and Distributed Systems*, pages 201–216, 2004.

[19] Helger Lipmaa. Aes ciphers: speed. Online at http://www.adastral.ucl.ac.uk/~helger/research/aes/rijndael.html.

[20] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2006.

[21] Len Sassaman, Bram Cohen, and Nick Mathewson. The pynchon gate: a secure method of pseudonymous mail retrieval. In *WPES*, pages 1–9, 2005.

[22] Radu Sion and Bogdan Carbunar. On the Practicality of Private Information Retrieval. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2007. Stony Brook Network Security and Applied Cryptography Lab Tech Report 2006-06.

[23] Shuhong Wang, Xuhua Ding, Robert H. Deng, and Feng Bao. Private information retrieval using trusted hardware. In *Proceedings of the European Symposium on Research in Computer Security ESORICS*, pages 49–64, 2006.