Building Castles out of Mud: Practical Access Pattern Privacy and Correctness on Untrusted Storage

Peter Williams
Stony Brook Network Security and Applied Cryptography Lab
Stony Brook, NY 11794-4400
petertw@cs.sunysb.edu

Radu Sion
Stony Brook Network Security and Applied Cryptography Lab
Stony Brook, NY 11794-4400
sion@cs.sunysb.edu

Bogdan Carbunar
Motorola Labs
1295 E. Algonquin Rd. IL05
Schaumburg, IL 60195
carbunar@motorola.com

ABSTRACT

We introduce a new practical mechanism for remote data storage with efficient access pattern privacy and correctness. A storage client can deploy this mechanism to issue encrypted reads, writes, and inserts to a potentially curious and malicious storage service provider, without revealing information or access patterns. The provider is unable to establish any correlation between successive accesses, or even to distinguish between a read and a write. Moreover, the client is provided with strong correctness assurances for its operations – illicit provider behavior does not go undetected. We built a first practical system – orders of magnitude faster than existing implementations – that can execute over several queries per second on 1Tbyte+ databases with *full computational privacy* and *correctness*.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software

General Terms

Security

Keywords

Data Outsourcing, Private Information Retrieval

1. INTRODUCTION

As networked storage architectures become prevalent – e.g., networked file systems and online relational databases in sensitive infrastructures 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. Such 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.

CCS'08, October 27–31, 2008, Alexandria, Virginia, USA.

Moreover, today, the remote servers are increasingly 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 [10]. However, 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 [1] and Xdrive [2]. 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 privacy. 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 information is still leaked through the access pattern of encrypted data. For example, consider an adversarial storage provider that determines 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 software. The adversary can then correlate plaintext keywords, identified by their position in the index, to documents, by observing which locations in the encrypted index are updated when a new encrypted document is uploaded. In general, it is difficult to bound the amount of information leaked by access patterns.

In existing work, one proposed approach for ensuring client access pattern privacy (and confidentiality) tackles the case of a single-owner model. Specifically, a service provider 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) [14]. For simplicity, in the following we will use the term ORAM to refer to any such outsourced data technique.

One of the main drawbacks of existing ORAM techniques is their overall time complexity. Specifically, in real-world setups ORAM [14] yields execution times of hundreds to thousands of seconds per single data access.

This Contribution. In this paper we propose to build on the work of Williams et al.[19] to introduce an efficient ORAM protocol with significantly reduced communication and computation complexities. Our protocol uses the ORAM-based [14] pyramid-shaped database layout and reshuffling schedule employed in [19] which yielded a protocol with a complexity of $O(\log^2 n)$ in the presence of $O(\sqrt{n})$ client working memory, for a database sized n. Here however, we deploy a new construction and more sophisticated reshuffling protocol, to significantly reduce both the computational complexity (to $O(\log n \log\log n)$) and the server storage overheads (to $O(n)$) – yielding a comparatively fast and *practical* oblivious data access protocol.

Efficiency. One of the main intuitions is to store each pyramid level as an encrypted hashtable and an encrypted Bloom filter (indexing elements in the hashtable). The Bloom filter allows the client to privately and efficiently – no linear scanning of $O(\log n)$ fake block buckets for each stored block to hide the success of each level query as in previous ORAMs – identify the level where an item of interest is stored, which is then retrieved from the corresponding hashtable. Less server-side storage is required ($O(n)$ instead of $O(n \log n)$), thus both increasing throughput and reducing required server-side storage by an order of magnitude.

Privacy. The approach guarantees client access pattern privacy, since the same operations are performed at all pyramid levels, in the same sequence for any item of interest. The use of the encrypted Bloom filters allows the client to query an item directly at each level without revealing the success, instead of relying on a series of $O(\log n)$ fake blocks for each stored block to hide the success of each level query. Our contributions consist also of a new reshuffling algorithm that obliviously builds and maintains the encrypted Bloom filters and of a more efficient oblivious merge-and-scramble.

Correctness. Moreover, authenticated per-level integrity constructs provide clients with *correctness* assurances at little or no additional cost, specifically ensuring that illicit server behavior (e.g., alterations) does not go undetected.

System. We built a system capable of executing several queries per second on 1TByte+ databases with full computational access privacy and correctness assurances. To the best of our knowledge, this is the first system in existence that offers these assurances at a practical throughput.

Moreover, our ORAM protocol is well suited for deployment on constrained hardware such as SCPUs. We propose its deployment on existing secure hardware (IBM 4764 [3]) to implement Private Information Retrieval (Figure 1), and show that the achievable throughputs are practical and much higher than existing work. These results contribute key insights towards making PIR assurances truly practical.

2. MODEL

Deployment. We consider the following concise yet representative interaction model. Sensitive data is placed by a client on a data server. Later, the client 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 with correctness assurances, while revealing a minimal amount of information (preferably none) to the (curious and possibly malicious) server. We describe the protocols from the perspective of the client, who will implement two privacy-enhanced primitives: read(id), and write(id, $newvalue$). The (un-trusted) server need not be aware of the protocol, but rather just provide traditional store/retrieve primitives.

Adversary. The adversarial setting considered through Section 4 assumes a storage provider that is *curious and possibly malicious*. Not only does it desire to illicitly gain information about the stored data, but it could also attempt to cause data alterations while remaining undetected. We prove that clients will detect any tampering performed by the server, before the tampering can affect the client's behavior or cause any data leaks. We do not consider timing attacks, noting that any implementation can be turned into a timing-attack free implementation without affecting the running time complexity. We also do not address direct denial of service behavior.

Cryptography. We require three cryptographic primitives with all the associated semantic security [13] properties: (i) a secure, collision-free 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

3.1 Oblivious RAM

Oblivious RAM [14] provides access pattern privacy to clients (or software processes) accessing a remote database (or RAM), requiring only logarithmic storage at the client. The amortized communication and computational complexities are $O(log^3 n)$. Due to a large hidden constant factor, the ORAM authors offer an alternate solution with computational complexity of $O(log^4 n)$, that is more efficient for all currently plausible database sizes.

In ORAM, 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, as a pyramid. 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.

ORAM Reads. To obtain the value of block id, the client must perform a read query in a manner that maintains two invariants: (i) it never reveals which level the desired block is at, and (ii) it never looks 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 re-encrypts the query result with a different nonce and places it in the *top* level. This ensures that when it 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. The top level quickly fills up; how to dump the top level into the one below is described later.

ORAM Writes. 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.

Level Overflow. Once a level is full, it is emptied into the level below. This second level is then re-encrypted and re-ordered, according to a new hash function. Thus, accesses to this new generation of the second level will hence-forth be completely independent of any previous accesses. Each level overflows 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 re-order 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 re-order process fills all partially empty buckets to the top with *fake* blocks. Recall that since every block is encrypted with a semanticly secure encryption function, the adversary cannot distinguish between fake and real blocks.

ORAM 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. In practice, implementations have a computational cost of $O(\log^4(n))$ as discussed above.

In [19] Williams et al. introduced an ORAM-variant with a cost of $O(\log^2 n)$ when $O(\sqrt{n})$ client storage is available. In their work, the assumed client storage is used to speed up the reshuffle process by taking advantage of the predictable nature of a merge sort on uniform random data. In this work we build on their result.

3.2 Private Information Retrieval

Another set of existing mechanisms handle access pattern privacy (but *not data confidentiality*) in the presence of *multiple clients*. Private Information Retrieval (PIR) [8] protocols aim to allow (arbitrary, multiple) clients to retrieve information from public or private databases, without revealing to the database servers which records are retrieved.

In initial results, Chor et al. [8] proved that in an information theoretic setting, any single-server solution requires $\Omega(n)$ bits of communication. PIR schemes with only sub-linear communication overheads, such as [8], require multiple non-communicating servers to hold replicated copies of the data. When the information theoretic guarantee is relaxed single-server solutions with better complexities exist; an excellent survey of PIR can be found online [11, 12].

Recently, Sion et al. showed [17] that due to computation costs, use of existing non-trivial single-server PIR protocols on current hardware is still orders of magnitude more time-consuming than trivially transferring the entire database.

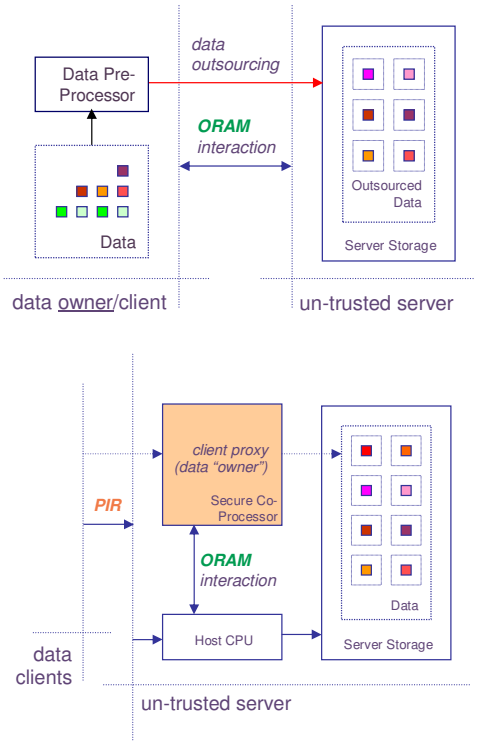

Figure 1: (top) Simple ORAM Protocol between a client and a server. (bottom) A trusted server-side client proxy can be used to build a PIR interface on top of ORAM assurances.

3.3 Secure Hardware-aided PIR

The recent advent of tamper-resistant, general-purpose trustworthy hardware such as the IBM 4764 Secure Co-Processor [3] has opened the door to efficiently deploying ORAM privacy 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.

Asonov was the first to introduce [4] 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 portions of the database on every request, indicating a computational complexity cost of $O(n)$, where n is the size of the database.

An ORAM-based PIR mechanism is introduced by Iliev and Smith [15], who deploy secure hardware to achieve a cost of $O(\sqrt{n}\log n)$. This is better than the poly-logarithmic complexity granted by ORAM for the small database sizes they consider. This work is notable as one of the first full ORAM-based PIR setups. Figure 1 summarizes the interaction between the client and server in ORAM, and how to turn an ORAM implementation into a PIR implementation using a Secure CPU.

An improved ORAM-based PIR mechanism with $O(n/k)$ cost is introduced in [18], 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 an improvement, this result is not always practical since the total database size n often remains much larger than the secure hardware size k. For $k = \sqrt{n}$ (as assumed in this paper), this mechanism yields an $O(\sqrt{n})$ complexity (significantly greater than $O(\log \log n \log n)$ for practical values of n).

4. A SOLUTION

In previous work [19] Williams et al. achieved a complexity of $O(\log^2 n)$ in a protocol offering *access privacy* but *no correctness* assurances. Here we build on their result by deploying a new construction and more sophisticated reshuffling protocol, to significantly reduce both the computational complexity and the storage overheads to only $O(\log n \log \log n)$ (amortized per-query), under the same assumption of $O(\sqrt{n})$ temporary client storage, while also endowing the protocol with correctness assurances.

4.1 Overview

Similar to ORAM (see Section 3.1 for more details), data is organized into $\log(n)$ levels, pyramid-like. Level i consists of up to 4^i items, stored on the server as label-value pairs. These pairs can be stored and retrieved in $O(1)$ time if the storage provider implements a suitable hash table [16]. This differs from ORAM, which stores an item at level i using a keyed hash function to determine its storage bucket (of size $O(\log n)$, to allow for hash collisions) within the level. The use of *fixed-sized hash buckets in ORAM instead of a simple hash table adds a $O(\log n)$ storage overhead multiplier, and slows down query processing*, but the buckets are necessary; otherwise queries to a hash table could reveal whether the item was found at this level.

Here *we avoid the overhead of using buckets* to mask the query result by using Bloom filters [7] (constructed to be collision-free). Before attempting to query for an item that might not be at the current level, a per-level Bloom filter is queried first. The bits of the Bloom filter are encrypted, hiding the result of the query. If the Bloom filter indicates that the item is *not* at this level, we query the level for a unique fake item instead and continue with the next level. Once we eventually find the desired item (at a future level) – it will be moved into the root tree node – above the levels where it was searched for before (as in ORAM). This ensures that the same item will never be queried for in that instantiation of the Bloom filter again (as now it will be found higher in the pyramid, or a reshuffle would have been triggered).

Insight One: Faster Lookup. Thus one key insight in our mechanism is that we can construct an encrypted Bloom filter to perform set membership tests, without revealing the success of our query. Additionally, we design a novel construction procedure that assembles the encrypted Bloom filter without revealing any correlation between scanned items and associated Bloom filter positions. The final benefit of using encrypted Bloom filters is that all unique queries are computationally indistinguishable due to the nature of the keyed hash function used to index the filter. This allows us to modify ORAM with significant performance benefits, since we can avoid handling hash collisions, which add a $\log n$ factor in total database size as described above.

The notion of encrypting a Bloom filter has been studied previously, e.g. in [6]. However, here we use a novel construction that hides the construction process, the inputs, and the results of the Bloom filter.

Insight Two: Correctness. Moreover, we deploy a set of authenticated, per-level integrity constructs to provide clients with *correctness* assurances at minimal additional cost. We specifically ensure that illicit server behavior (e.g., alterations) does not go undetected.

We now detail these components.

4.2 Query Processing

A query consists of a read or write request for a data item. These items are kept at the storage provider at a particular level; part of the client's job is to determine which level the item is at without revealing this to the server. Algorithm 1 shows the pseudo-code of this operation.

To process a query, the client first downloads and scans the server-stored item cache (line 10) then proceeds to search each level, starting at the top (line 11). A labeling function consisting of a hash of the item ID with several level parameters ($fakeAccessCtr(level)$ and $Gen(level)$) generates the unique label by which the client can find the item at a particular level, if the item is indeed there. $fakeAccessCtr(level)$ represents the number of accesses to $level$ since the last reshuffle, and $Gen(level)$ represents the number of times $level$ has been reshuffled. The use of $fakeAccessCtr$ (line 12) ensures that successive queries request unique fake items, which the client knows are stored on the server. The use of $Gen(level)$ ensures that items on every subsequent reshuffle of a level have unique labels. Both functions are computed from the total number of accesses to the system thus far.

The search of any one level requires $O(1)$ time. If at the ith level the item has not already been found (in the cache or a previous level) (line 14) the client first computes the label under which the item would be stored in the ith level Bloom filter (line 15). It then retrieves the encrypted bits corresponding to that label from the server-stored Bloom filter (line 16). If the decrypted bits are all 1 (line 17), the client has found the item at the ith level. It then computes the label under which the item is stored in the level (referenced by its hashtable) (line 18) and asks the server to remove and return the corresponding item (line 19). If at least one decrypted Bloom filter bit is 0 (line 21), the client instead performs the same operation using a *fake* label (built at line 13), known to be stored at the server (line 20).

Once the client has found the item (line 21), it proceeds by seeking fake items on the subsequent levels. This avoids revealing the level that answered the query, which would provide a correlation between queries. The client first searches for a fake label (line 22) in the Bloom filter at the ith level (line 23), then asks the server to retrieve and remove a fake item from the level (line 24).

Note that the client queries the remotely-stored encrypted Bloom filter by requesting the encrypted values of positions indicated by the label function. While this reveals the requested Bloom filter positions to the remote server, *nothing is lost* as we prevent correlation by guaranteeing that any Bloom filter is only ever queried for any particular item once. Since the positions in the filter are each encrypted, the server never learns the result of the Bloom filter query.

4.3 Access Privacy

The client achieves access pattern privacy by maintaining two conditions. First, *no item is ever queried twice using the same label.* This is achieved by removing the item, once it is found, and placing it in the item cache. Thus, on fu-

Algorithm 1 Query answering overview.

```
1.query(x : id)
2.  server : Server; #server stub
4.  bits : int[]; #bit values in Bloom filter
5.  label, fakeLabel : int[]; #search labels
6.  fakeAccCtr : int[]; #per level access counter
7.  found : bool;
8.  K : int[]; # secret key
9.  v : Object; # value for name x
10. found, v := scanServerItemCache(x);
11. for (i := 1; i < log_4 n; i++) do
12.     fakeAccCtr(i)++;
13.     fakeLabel := hash(i, "data", Gen(i), fakeAccCtr(i), K);
14.     if(found = false) do
15.         label := hash(i, "BF", Gen(i), "x", K);
16.         bits := server.getBloomFilter(i, label);
17.         if(decrypt(bits) = "11..1") do
18.             label := hash(i, "data", Gen(i), "x", K);
19.             v := server.getNRemove(label); found := true;
20.         else server.getNRemove(fakeLabel) fi;
21.     else
22.         label := hash(i, "BF", Gen(i), fakeAccCtr(i), K);
23.         server.getBloomFilter(i, label);
24.         server.getNRemove(fakeLabel);
25.     fi
26. itemCache.append(x, v);
27. return (x, v);
28.end.
```

ture queries the client will locate it in the item cache before repeating a label request; fakes will be substituted on the lower levels. As items propagate out of the item cache (described in Section 4.4), the label functions are updated, so that the item has a different label by the time it makes it back down to a particular level.

Second, *the access patterns must appear indistinguishable* from random no matter where the item is located. A set of fake items is used to guarantee this: if the Bloom filter returns negative, indicating that the item is not stored at this level, a fake item from this level is retrieved instead.

On every single query, the server observes the same pattern. The client first scans the item cache, then queries a random value (chosen uniformly randomly, independently from all other information available to the server) from the level 1 encrypted Bloom filter – never queried by the client before. The server can observe the positions in the Bloom filter accessed, but it cannot observe whether each position is set to 1 or 0. The server then observes the client retrieve and delete one item from the level 1 hash table – never retrieved by the client before. This identical pattern of a random Bloom filter lookup followed by a random label-value retrieval and deletion continues through each level. Finally, the client appends a (semantically secure) encrypted value to the item cache.

Success or failure at each level is not revealed – the server cannot distinguish queries to fake entries in the Bloom filter from queries to real items in the filter, and the server cannot distinguish either of those from real items that are not in the filter. Additionally, the server cannot distinguish requests to real items from requests to fake items from the hash table, since the secure hash function used is non-invertible.

Since there are $\log_4 n$ levels, and a constant amount of data transfer and computation is exercised on each level by every query, the online cost per query is $O(\log n)$ (measured in computation or transfer of words). Level reshuffling, described in the next section, will add an offline amortized cost per query of $O(\log n \log\log n)$.

We now fill in the missing pieces: how to empty the item cache when it becomes full, and how to build the levels and the Bloom filters without revealing any information the server can correlate to retrievals.

4.4 Handling Level Overflows: Reshuffle

The construction of the initial database structure is explained by the process of emptying the item cache: items are inserted into the item cache, which then overflows into the lower levels. Similar to ORAM, when the item cache is emptied, the contents are poured into level 1. In that process, level 1 and its new contents are reshuffled according to new label functions, removing any correlations between past and future lookups. When level 1 becomes full, it is poured into level 2, and so forth. Thus, the reshuffle process empties one level $i-1$ into the level i below it, which is four times as large as level $i-1$. Level i is then scrambled, hiding the correlation with the items' previous levels. A new Bloom filter for the lower level is constructed – even items which happened to be at level i anytime in the past are now identified by a new unique label.

Let m be the size of the new level ($m \leq 4^i$). Let h be the number of hash functions used to generate a Bloom filter. Let $k_1, ..k_h$ be the client's secret keys used to generate the Bloom filters. Let b denote the number of bits in the Bloom filter BF at level i. Let W be a working set stored on the server. Let L be a list of $O(m)$ entries, stored on the server. Let T be a $\sqrt{m}$ integer array stored at the client. Let Bkt be a server-hosted list of $O(\sqrt{m})$ buckets, of $\sqrt{m}$ entries each. Initially, W, L, T and Bkt are empty and all the bits in the server-stored BF are set to 0.

In the next steps (steps 2 through 7 in the detailed description below, and illustrated in Figure 4.4) we build the encrypted Bloom filter without revealing the positions set in the Bloom filter. This is accomplished by scanning all items in the level, creating a list of what positions must be set in the Bloom filter to add each item, and storing this list encrypted on the server (step 2). To turn the list into a proper Bloom filter bit array, it will be sorted with a bucket sort – with $\sqrt{m}$ buckets of size $\sqrt{m}$, so that any bucket fits in private storage. To keep the buckets indistinguishable, we ensure they will all have the same size. Next (step 3) we calculate the size of each bucket, by scanning the list of Bloom filter positions, incrementing the appropriate bucket size tally for each position. In step 4 we add fake positions that will end up in those buckets that are lacking, according to the above (step 3) tally. Each bucket corresponds to a fixed range of positions in the final filter, so $j\sqrt{m}$ is a position that will wind up in bucket j. At this point the server will be able to identify the fake positions, since they are all at the end of the list. We then (step 5) scramble the list of positions to destroy all correlation between items and positions, and hide the fakes. In step 6 we move the scrambled positions into their buckets. Step 7 constructs the final Bloom filter, building a piece from each bucket. Steps 8 and 9 place the items in the new level while eliminating any correlation between the old and the new level structures.

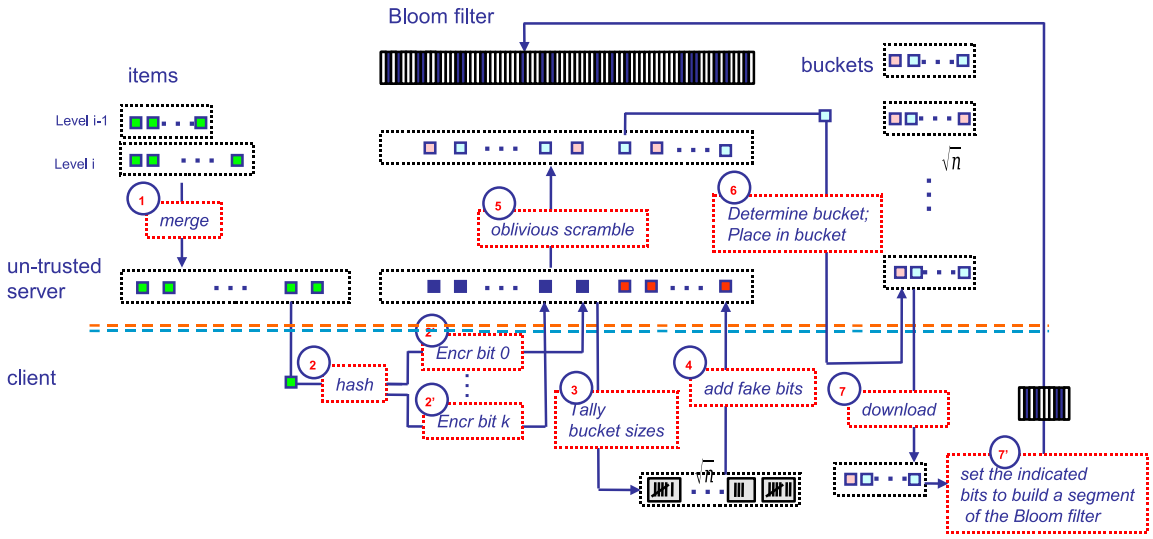

Figure 2: Level reshuffle: Bloom filter construction (Steps 2 - 7)

The overflow process, performed by the client to pour level $i-1$ (or the item cache) into level i proceeds as follows (see Figure 4.4 for an illustration).

1. **Merge levels.** Move all items from level $i-1$ and level i into W, a working buffer on the remote server. Discard the Bloom filters attached to both levels.

2. **Build a list representation of the new Bloom filter.** Increment the $Gen(Li)$ value. Read each item $x \in W$ exactly once, and for each compute its h bit-positions $p_j(x) = hash(Gen(Li)||x.id||i||k_j)\ mod\ b$, $j = 1..h$, in the new Bloom filter. Encrypt each $p_j(x)$ separately and store all $E(p_j(x))$ values on the server-side list L. This step takes $O(m)$ time, and $O(1)$ private (client-side) storage. *Costs here and in the following steps are expressed in terms of $m = 4^i$ being the size of the current level.*

3. **Tally Bloom filter positions to determine future bucket sizes.** Read each entry of L (the list of encrypted Bloom filter positions prepared in the previous step) exactly once. At the client side, for each entry $E(p) \in L$, let $idx(p)$ be the $\log m/2$ most significant bits of p. Then, do $T[idx(p)]++$. This step allows the client to compute the number of bit-positions of L that will later (Step 6) end up in the Bkt structure. Effectively, the client builds a tally in local storage calculating the future size of each of the $\sqrt{m}$ buckets that will be built on the server in the step 6 bucket sort. We use $\sqrt{m}$ buckets of size $\sqrt{m}$ so that each bucket will fit in private storage in step 7, and the tally built here, with one counter per bucket, also fits in private storage. The step requires $O(m)$ time and $O(\sqrt{m})$ private storage. (To avoid redundant scans, this step can be merged with the previous).

4. **Add fake bits to make the bucket sizes equivalent.** The local tally from step 2 indicates the size of the largest bucket. We scan the tally, adding fake encrypted positions to the server-side list of encrypted positions as we go, so that all the buckets will have the same size as the largest bucket after the step 5 bucket sort. To add a fake position that will correspond to bucket j, the position $j\sqrt{m}$ is added to the list. (A simple balls and bins result predicts that the $\sqrt{m}$-sized buckets all have similar sizes, already; the number of fakes to add is small compared to the number of real items). Let max be the index of T such that $T[max] = max_{j=1}^{\sqrt{m}} T[j]$. For each $j = 1..\sqrt{m}$ generate $T[max]-T[j]$ fake values v_l such that the $\log m/2$ most significant bits of each v_l are equal to j. Store the encrypted, $E(v_l||\text{“}fake\text{''})$ value in L. This operation ensures that all the buckets of Bkt will store the same number of elements. The bucket size tally is discarded after this step. This step requires $O(m)$ time.

5. **Obliviously scramble the list of Bloom filter positions.** The encrypted indexes (the bit-positions of L, including the fakes) are scrambled, according to our Oblivious Merge Scramble Algorithm, which destroys all correlation between the old positions and the resulting positions, which are a new uniform random permutation. The new list L stores the scrambled values. The algorithm requires $O(m \log\log m)$ time and $O(\sqrt{m})$ private storage.

6. **Bucket-sort the list of Bloom filter positions.** For each $E(p) \in L$, let $idx(p)$ be the $\log m/2$ most significant bits of p. Then, do $Bkt[idx(p)].add(E(p))$. Here the Bloom filter's scrambled, encrypted positions are bucket-sorted. The client retrieves each bit index, decrypts it to read it, and writes the encrypted value back to the bucket on the server corresponding to the $\frac{\log m}{2}$ most significant bits of the position. The bucket sort allows us to construct the encrypted Bloom filter in the next step without revealing to the server which

bits are set: if we were to simply scan the entire list of positions setting the corresponding bits to true, the server would observe the bit flips in our encrypted array and learn what positions are set. The bucket sort groups related positions together so that we can build the Bloom filter from left to right in a single pass. This step requires $O(m)$ time.

7. **Construct Bloom filter.** For each $j = 1..Bkt.size$, download $Bkt[j]$. Note that the size of $Bkt[j]$ is $\sqrt{m}$. Let $BF[j\sqrt{m}]..BF[(j+1)\sqrt{m}]$ be the segment of the Bloom filter corresponding to $Bkt[j]$, where $BF[idx]$ denotes the idxth bit of BF. For each $E(p) \in Bkt[j]$, let x be the least significant $\log m/2$ bits of p. Do $BF[j\sqrt{m} + x] = 1$. Store $E(BF[j\sqrt{m}]..E(BF[(j+1)\sqrt{m}]$ on the server. Finally, store the oblivious Bloom filter of the working set W on the server.

   Here the client downloads each bucket (which conveniently fits into local storage). The bucket corresponds to a $\sqrt{m}$-sized segment of the final Bloom filter – all positions in this bucket refer to a bit in this segment of the filter. The bits corresponding to listed positions are set to true, with all other bits set to false, in the local copy. The client encrypts this Bloom filter segment and uploads it to the server. Observe that the server has no indication of how many bits are true in this segment (other than that it is limited by the bucket size), nor which are true. The Bloom filter is finished at the end of this step. This step requires $O(m)$ time, and $O(\sqrt{m})$ private storage.

8. **Scramble the items.** Finally, the client uses the Oblivious Merge Scramble Algorithm to scramble the actual items in the working buffer W. The Oblivious Merge Scramble requires $O(m \log\log m)$ time and $O(\sqrt{m})$ private storage.

9. **Add items back to level i.** Once scrambled, the items inserted under their new labels, according to the new labeling function for level i. For each item in $x \in W$ let $label(x) = hash("LiData||Gen(Li)||k)$. Insert the pair $(x, label(x))$ into the set of items stored at level i. Add m fake items to level i, so that a query that turns out not to be for this level will have an item to retrieve instead (most of these m fakes will be deleted by the query process before the next reshuffle).

Level $i-1$ is now empty, and level i now contains all the items that were in level $i-1$. If level i is now full, this is then repeated as level i is then dumped into level $i+1$ etc.

This procedure shows how level $i-1$ can be dumped into level i at a cost of $O(m \log\log m) = O(4^i \log i)$. Level $i-1$ is emptied once every 4^{i-1} queries, thus resulting in an amortized cost per query due to reshuffling of

$$\sum_{i=0}^{log_4 n} O(\frac{4^i \log i}{4^{i-1}}) = \sum_{i=0}^{log_4 n} O(\log i) = O(\log n \log\log n)$$

The Bloom filter bits retrieved to check an item will appear to be chosen uniformly random, and completely independently of each other; therefore, any bit pattern indicates nothing about the query or the success of the query. They are independent of the bucket sort write pattern, which is the only other piece that could be tied to it, since the bucket sort write pattern is the only access pattern that varies during the reshuffle. The bucket writes are all identical except for the order of the writes, which is uniform random because of the scramble. The scramble has no bearing on the Bloom filter access pattern, which is dependent only on the query and the current Bloom hash function. Therefore the Bloom filter construction process yields no information about the items to the server in the resulting Bloom filter.

The level reorder process results in a new level that has no correlation to the old level, since the new permutation is chosen uniformly randomly (Theorem 2). The scramble process itself reveals no information about the new or old permutations, since the scramble has the same access pattern in all instantiations.

4.5 Oblivious Scramble Algorithm

To complete step 5 above, we now describe an algorithm that performs an oblivious scramble on a array of size n, with $c\sqrt{n}$ local storage, in $O(n \log\log n)$ time with high probability. This is based on an algorithm by Williams et al. in [19], which scrambles an array obliviously in time $O(n \log n)$ by ways of a merge sort. Since our application only requires a scramble, and not a complete sort, we can improve the asymptotic complexity by merging multiple arrays at once.

Informally, the algorithm is still a merge sort, except a random number generator is used in place of a comparison, and multiple arrays are merged simultaneously. The array is recursively divided into segments, which are then scrambled together in groups. The time complexity of the algorithm is better than merge sort since multiple segments are merged together simultaneously. Randomly selecting from the remaining arrays avoids comparisons among the leading items in each array, so it is not a comparison sort.

The Oblivious Scramble Algorithm proceeds recursively as follows, starting with the remote array split into segments of size $s = 1$, a security parameter c, and an array to scramble of size n.

1. For segments sized s, allocate $\lceil\sqrt{n/s}\rceil$ buffers of size $c\sqrt{s}$, (requiring $c\sqrt{n}$ space total)
2. Split the array into groups of $\sqrt{n/s}$ segments.
3. For each of the $\frac{n/s}{\sqrt{n/s}} = \sqrt{n/s}$ groups:
   - Obliviously merge the segments in this group together into one new segment of size $(s)\sqrt{n/s} = \sqrt{ns}$, by performing the Oblivious Merge Step on the allocated buffers. The Oblivious Merge Step requires $c\sqrt{s}$ local working memory for each of the $\sqrt{n/s}$ buffers, for a total of $c\sqrt{n}$ working memory, and operates in $O(\sqrt{ns})$ time.
4. In the end there are $\sqrt{n/s}$ segments of size $\sqrt{ns}$.
5. Repeat.

One recursion of this algorithm requires a single pass across the level, costing $O(4^i)$ for level i. Each pass brings the total number of segments from n/s to $\sqrt{n/s}$, and we repeat until there is one segment left. After iteration p, the number of segments remaining will be $n^{1/2^p}$. There will be 2 segments left when $p = \log\log n$. Since it takes $\log\log n$ passes to go

from n to 2, and each pass involves a single read and write of the entire array, the total running time / communication complexity for running the oblivious scramble on level i is $O(4^i(\log\log 4^i + 1)) = O(4^i \log i) = O(n \log\log n)$.

We now describe the last remaining piece of the Oblivious Merge Scramble Algorithm, the Merge Step.

4.6 Oblivious Merge Step

The Oblivious Merge Step, whose pseudo-code is shown in Algorithm 2, takes r arrays of size n/r, and merges them randomly into a single array of size n, preserving the ordering among the input arrays in the output arrays: if an item a is before item b in original array i, it will also be before b in the final array.

The permutation is chosen uniformly randomly out of all permutations that preserve the ordering of the original input items. To ensure this, we will take n steps, choosing an item from the front of one of the r arrays at every step. The choice is biased since we choose each item without replacement randomly from the remaining items: if a particular array has a items left at step j, it has a $\frac{a}{n-j}$ chance of being chosen at this step.

The key to obliviousness is that we accomplish this random selection without affecting the actual access pattern of reading from the server. In [19] this is implemented for 2 arrays; we now extend this to merge r arrays. By simply reading the input evenly at a fixed rate, and outputting the items indicated by the random function, the uniform nature of the random function will cause the output rates to be very similar with high probability.

In other words, we maintain a series of caching queues that are fed at a certain rate. According to a random function, we remove items from the queues. By the nature of the random selection, with high probability the queues will never overflow from being dequeued too slowly, nor empty out from being dequeued too quickly, as shown in Theorem 1. Due to space requirements, the proofs are omitted from this version of the paper.

THEOREM 1. *The Oblivious Merge Scramble succeeds, with high probability: the chance that the queue buffers overflow or underrun is negligible w.r.t. the security parameter c.*

THEOREM 2. *The Oblivious Merge Scramble produces a permutation selected uniformly randomly from the set of all permutations.*

THEOREM 3. *The server learns nothing about the access pattern from a client running this protocol.*

4.7 Bloom Filter Parameters

We discuss here suitable choices of Bloom filter parameters. A Bloom filter containing z items has two parameters: y, the number of hash functions used (the number of bits set per item in the filter), and x, the number of bits in the Bloom filter, yielding the false positive rate $r = \left(1 - \left(1 - \frac{1}{x}\right)^{yz}\right)^y$.

Our Bloom filters are constrained by two important considerations. First, we need to minimize y, since this corresponds to the number of disk seeks required per lookup. Second, we need to guarantee that with high probability, there will be no false positives; i.e., r must be negligible to prevent a privacy leak.

Algorithm 2 Oblivious Merge Step

```
1.oblivious_merge_step(A_1[], ...A_r[])
2.  B : array[n]; #new remote destination buffer of size n
3.  s := 2c√(n/r); #size of local queues
4.  for (i := 1; i ≤ r; i++) do
5.      q_i := new queue[s];
6.      for (x := 1; x ≤ s/2; x++) do
7.          q_i.enqueue(decrypt(A_i.readNextItem()));
8.  #at this point each queue has s/2 items
9.  for (x := s/2; x ≤ n + s/2; x++) do
10.     if (x ≤ n) then
11.         for (i := 1; i ≤ r; i++) do
12.             q_i.enqueue(decrypt(A_i.readNextItem()));
13.     fi
14.     # now we have read r items; time to output r items
15.     for (i := 1; i ≤ r; i++) do
16.         v := randomlyChooseWhichArray();
17.         t := q_v.dequeue();
18.         B.writeNextItem(encryptWithNewNonce(t));
19.end.
```

Therefore, for any fixed acceptable error rate r, e.g., 2^{-64}, and for member count $z = m$ (level size), the trade-off between the Bloom filter size x and the bits set per item y must be optimized to balance online disk seeks for query answering, server storage used, and Bloom filter construction time. For disk-based storage of large databases, we find nearly optimal parameters by fixing $y \approx 5$, yielding a large, sparsely populated Bloom filter.

5. CORRECTNESS AND INTEGRITY

In this section we introduce a set of integrity constructs that endow the above solution with correctness assurances. Specifically, we would like to guarantee that any storage provider tampering behavior is detected. All of these constructs can be implemented efficiently, with few or almost no overheads: (i) Message Authentication Codes (MACs) are added for all the stored items and Bloom filters; (ii) unique version labels for each item in the data covered by the MAC are added to prevent any replay-type attack in which the server incorrectly replies with a previously MAC-signed message; (iii) incremental, collision-resistant commutative checksums are added to checksum the item sets contained in each level, to prevent the server from hiding or duplicating items during the level reshuffle process.

For (i) we require a MAC function, such that the computationally bounded adversary has no non-negligible ability to construct any message, MAC pair $(M, MAC(M))$ for an M that did not originate at the client. Every item uploaded to the server is protected by such a MAC. (ii) is straightforward. For (iii), besides the requirement of being collision-resistant, it should be easy for clients to maintain and update a checksum of a set. In particular, when adding or deleting items, it should be possible to to so incrementally, without recomputing the entire checksum value.

The *incremental hashing* paradigm of Bellare and Micciancio [5] can be used to construct exactly such a checksum. Fix any cryptographic hash function h (viewed as random oracle) and a large prime p. To hash a set $B = \{b_1, \ldots, b_l\}$, we

compute the product

$$H(B) := \prod_{i=1}^{l} h(b_i) \bmod p. \quad (1)$$

Note that this hash construction allows both for easy addition of item b (multiplication with $h(b)$) and removal of any b_i (multiplication by $(h(d_i))^{-1}$) without needing to recompute the hashes of all values $b_1, \ldots, b_l$.

It can be shown (in the random oracle model [13], proof is out of scope here) that it is computationally infeasible to find two sets that have the same checksum; the hash function (1) thus forms an easy and efficient way to authenticate the set of items in a level:

THEOREM 4. *If the discrete logarithm problem in $\mathbb{Z}_p^*$ is hard it is computationally infeasible to find two sets $A \neq B$ with $H(A) = H(B)$.*

THEOREM 5. *A client interface correctly implementing the above integrity constructs will detect all incorrect server responses before they reveal any part of the access pattern, or return an incorrect answer to the user of the interface.*

6. PERFORMANCE

We implemented a prototype of the mechanisms discussed above. This allowed us a unique insight in the boundary between theoretical complexity and runtimes. We faced numerous challenges to allow TB(Terabyte) - level, multi-disk data handling, including implementing efficient TB - sized hashtables, multi-threading, fast packet queueing and data request handling, tweaking TCP/IP sockets to handle efficiently without delays, as well as generally caring about every spent microsecond.

We chose Java as an initial platform and compiled using the Sun JDK 1.6.0_05. The testing environment included a mix of 4 of the following drive types: Seagate Barracuda 7200.11 SATA 3Gb/s 1TB, 7200 RPM, 105 MB/s sustained data rate, 4.16ms average seek latency and 32 MB cache, and Western Digital Caviar SE16 3 Gb/s, 320GB, 7200 RPM, 122 MB/s sustained data rate, 4.2ms average latency and 16MB cache. The machines involved were running Intel(R) Pentium(R) 4 CPUs at 3.00GHz, with 2MB of L2 cache, 2GB RAM, and Linux Redhat Fedora Core 8, kernel 2.6.23.1 with Ext4 file system support enabled. To evaluate the suitability of the mechanisms for different network types we modulated different network delays and also ran the suite in a general purpose 100MBps CDMA ethernet network.

Growing Database. Figure 3 shows observed query response times over a database growing in size (by write queries) from 10 to 650 MBytes. A majority of queries require a few hundred milliseconds to run. Points above the average represent reshuffles of large levels. Large levels require more time to reshuffle, but are reshuffled less frequently. The triples of points show that each successive reshuffle of a level requires more time as the level grows; after the fourth reshuffle, the level is empty. The bands below the average query time reflect that after a large reshuffle, most of the levels are empty, decreasing the number of levels that must be examined in the following queries.

Impact of Network Latency. Figure 4 explores the impact of network latency on response time (simulated using *sleep*; the granularity of that primitive affects the accuracy of the points at 0ms and 5ms). A strong dependence on network round trip time can be observed, due to the fact that round trip costs are paid several times during the online phase of each query.

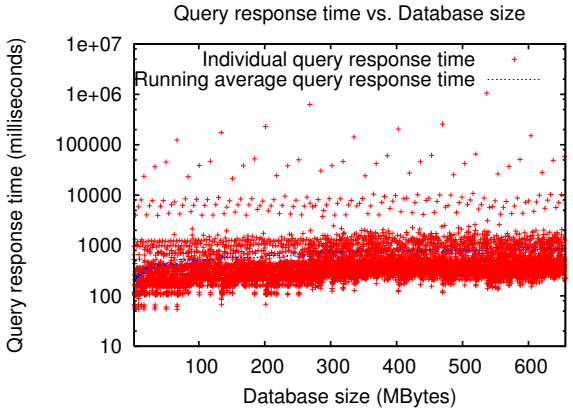

Figure 3: A database is built from 64KB items using a Bloom filter collision rate of 2^{-35} with network RTTs of 25ms (logscale on y axis).

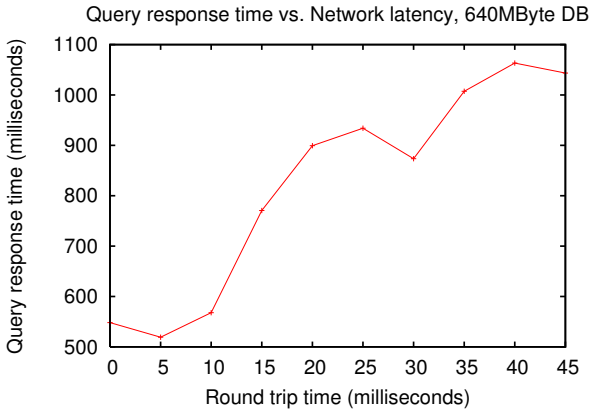

Figure 4: We issue 1000 read queries for items from the previously constructed 640MB database, averaging query response time .

Impact of Database Size. Figure 5 explores the impact of database size on response time in a controlled low-latency network setting (localhost network < .1ms latency). Here the databases are not grown as in figure 3, but rather queries are issued on a pre-constructed database of the indicated size. Multiple hard disks are used to incur disk seek cost in parallel. The behavior validates the $O(\log n \log \log n)$ complexity and looks almost logarithmic for the considered database sizes. For a 1 TByte database, over $2\frac{1}{4}$ queries per second can be performed.

Improvements. Our implementation suffers from two bottlenecks that are not inherent to the protocol. First, the *Java implementation* of the cryptographic primitives is significantly slower than what can be achieved on the hardware used. This slow-down is most prevalent during the reshuffle process, in which we encountered an unexpected CPU bottleneck instead of the expected I/O bottleneck. Second,

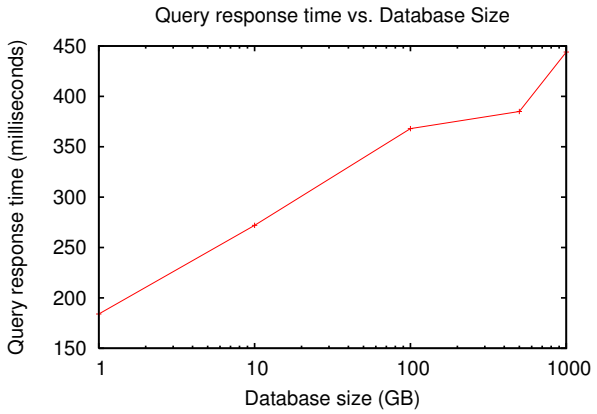

Figure 5: The impact of database size on query response time in a controlled, low-latency network setting on a pre-constructed database (logscale on x axis).

our implementation runs queries *synchronously*. The highly interactive nature of our query process, in which the output from one level is needed before the query to the next level can be composed, requires that we pay the network round-trip delay multiple times per query, as illustrated in Figure 4.A higher query throughput can be achieved by running multiple queries simultaneously, since the network delay does not represent a resource bottleneck. The current implementation does not yet support simultaneous queries, however this can be achieved (at least for those queries that don't trigger reshuffles) without affecting the rest of the protocol. Additionally, disk seek times will be mitigated in a parallel implementation, for the same reason, if there are multiple hard disks on the provider, since the disk seek penalty can be paid simultaneously across different disks.

7. CONCLUSIONS

In this paper we introduce a first practical oblivious data access protocol with correctness. The key insights lie in new constructions and sophisticated reshuffling protocols that yield practical computational complexity (to $O(\log n \log\log n)$) and storage overheads (to $O(n)$). We also introduce a first practical implementation that allows a throughput of several queries per second on 1Tbyte+ databases, with *full computational privacy* and *correctness*, orders of magnitude faster than existing approaches.

8. ACKNOWLEDGEMENTS

We would like to thank our anonymous reviewers, who offered helpful insights. 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.

9. REFERENCES

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

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

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

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

[5] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *Proceedings of EuroCrypt*, 1997.

[6] Steven M. Bellovin and William R. Cheswick. Privacy-enhanced searches using encrypted bloom filters. Technical report, Columbia University, 2004.

[7] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

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

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

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

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

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

[15] 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.

[16] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[17] 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.

[18] 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.

[19] Peter Williams and Radu Sion. Usable Private Information Retrieval. In *Proceedings of the 2008 Network and Distributed System Security (NDSS) Symposium*, 2008.