

Practical Oblivious Outsourced Storage

PETER WILLIAMS

RADU SION

MIROSLAVA SOTAKOVA

Stony Brook University

In this paper we introduce 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 of temporary storage (enough to store $O(\sqrt{n \log n})$ items and IDs, where n is the number of items in the database), we can achieve access pattern privacy with computational complexity 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, 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.

Categories and Subject Descriptors: H.2.7 [**Database Management**]: Database Administration

General Terms: Security, integrity, and protection

Additional Key Words and Phrases: private information retrieval, access privacy, oblivious RAM

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 infrastructures such as email and storage portals, libraries, and 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 cost [Gartner, Inc. 1999]. Moreover,

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 Cryptography Group, CEWIT, and the Stony Brook Office of the Vice President for Research.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 20YY ACM 0000-0000/20YY/0000-0001 \$5.00

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 FilesAnywhere.com, Gmail, Yahoo! Mail, and MSN Hotmail [Hild and Mitchell 2004]. 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 [Scribner 2007].

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 cooperating, non-malicious servers. As a first line of defense, for ensuring confidentiality, all 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. However, it 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 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 software 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) [Chor et al. 1995]. PIR protocols aim to allow clients to retrieve information from public or private databases, without revealing to the database servers which record is retrieved. However, we showed [Sion and Carbunar 2007] that deployment of existing number-theory based single-server PIR protocols on real hardware 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.

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) [Goldreich and Ostrovsky 1996]. For simplicity, in the following we use the term ORAM to refer to any such outsourced data technique.

In this paper we first introduce an efficient ORAM protocol with significantly reduced computation complexity ($O(\log^2 n)$ vs. $O(\log^4 n)$ for [Goldreich and Ostrovsky 1996]). In Section 6, we propose its deployment on existing secure hardware (IBM 4764 [IBM Corp. 2008]) and show that the achievable throughputs are practical and orders of magnitude higher than existing work.

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, integrity, and authenticity 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 the primitives: $\text{read}(id)$, and $\text{write}(id, \text{newvalue})$. 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).

The algorithm must be implemented in a manner to avoid timing attacks and other side channels, which otherwise would have the potential to leak some information. We assume that the implementation is built in such a manner. We note that any implementation can be turned into a timing-attack-free implementation by waiting longer on execution paths determined by secret information, such that for a given section of code, the length of all execution paths matches the length of the longest possible execution path. We also note that such a transformation 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. Finally, we 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 and Notation. We require three cryptographic primitives with all the associated semantic security [Goldreich 2001] properties: (i) a family of secure hash functions, such that the output of a function from the family is indistinguishable from the output of a random function, (ii) an encryption function that generates different ciphertexts over multiple encryptions of the same item, such that a computationally bounded adversary has negligible advantage at determining whether a pair of encrypted items of the same length represent the same or unique items, and (iii) a pseudorandom number generator whose output is indistinguishable from a uniform random distribution over the output space. In fact, assuming (i) and (ii) is sufficient, since pseudorandom number generators can be built from hash functions. $\log()$ denotes the natural logarithm unless specified otherwise.

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 [Chor et al. 1995]. In initial results, it was shown [Chor et al. 1995] 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, for a database of size n . To avoid this overhead, if multiple non-communicating databases can hold replicated copies of the data, PIR schemes with only sublinear communication overheads are shown to exist [Chor et al. 1995]. For example, [Sassaman et al. 2005] 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 2007] 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 [Gasarch 2010; 2004].

It is worth noting that Asonov was the first to introduce [Asonov 2004] a PIR scheme that uses a secure CPU to provide (an apparent) $O(\log n)$ 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. Nevertheless, this technique of using secure hardware to transform an *access pattern privacy* algorithm into a *private information retrieval* implementation, by using secure hardware as a trusted party to run the access pattern privacy algorithm, is employed by all the following PIR schemes.

Thus, the remaining secure-CPU based PIR schemes we consider, including our own discussed in Section 6, require only $O(\log n)$ communication between the client and secure CPU. This is a tight lower bound, as $\log(n)$ bits are required to specify the desired record. The computational cost of each scheme varies, and this determines the practicality, which is our main interest.

ORAM. Oblivious RAM [Goldreich and Ostrovsky 1996] provides access pattern privacy on a database, requiring only logarithmic private storage. The amortized computational complexity is $O(\log^3 n)$, 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 2004], who deploy secure hardware to obtain PIR at a cost of $O(\sqrt{n} \log n)$ using \sqrt{n} storage, and additionally allow a storage/performance tradeoff. 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 implementations, albeit with lower query throughputs.

A PIR mechanism with $O(n/k)$ cost is introduced in [Wang et al. 2006], where n is the database size and k is the amount of private storage (measured in items and indices). 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 private 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 [Yang et al. 2008] the authors introduced a new mechanism performing queries at an amortized cost of $O(\sqrt{n})$, requiring only constant private storage. After publication, however, the following problem was discovered: achieving this performance requires securely shuffling the entire database in $O(n)$ time with only constant private storage, which is currently not known to be possible.

We are exploring new constructions in recent results [Williams et al. 2008] that significantly reduce the required server storage (by eliminating the use of fake items) and potentially speed up operations by another order of magnitude. The idea is to use new data structures to obliviously locate a stored item, instead of iteratively scanning for the item, as in ORAM and the solution presented here. We also employ cryptographic checksums to prevent attacks by a fully malicious adversary.

In this paper we introduce a solution with only $O(\log^2 n)$ amortized overhead, in the presence of enough temporary client storage to store $4c\sqrt{n\log n}$ items and identifiers, where c is an independent security parameter. We show this to be a solution that can be implemented efficiently over large data sets.

Oblivious RAM Overview.

Since the proposed protocol is based on ORAM [Goldreich and Ostrovsky 1996], a brief summary of the operation of ORAM follows. The database is considered a set of n encrypted blocks and supported operations are $\text{read}(id)$, and $\text{write}(id, \text{newvalue})$. For simplicity, we assume that $n = 4^j$ for $j \in \mathbb{N}$. The data is organized into $\log_4(n)$ levels, pyramid-like. Level i contains up to 4^i blocks. In the execution, hash functions are used to assign each block to one of the 4^i buckets at this level, in a way indistinguishable from the case where each block is placed into a bucket chosen independently and uniformly at random from all buckets at the same level. Therefore, we need families of hash functions \mathcal{H}_i for $i \in \{1, \dots, \log_4 n\}$, such that for $h \in \mathcal{H}_i$, $h : \mathcal{B} \rightarrow 4^i$, where \mathcal{B} denotes the set of all block identifiers. Due to hash collisions each bucket may contain from 0 to $\lambda \log n$ blocks, for a security parameter $\lambda > 0$. We discuss λ in Section 4.6, noting that in most cases e is a good choice.

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 retrieve a given block from the same spot twice. 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 (a block) 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. Figure 1 illustrates this process. Note that the top level will immediately fill up; the process to dump the top level into the one below is described later.

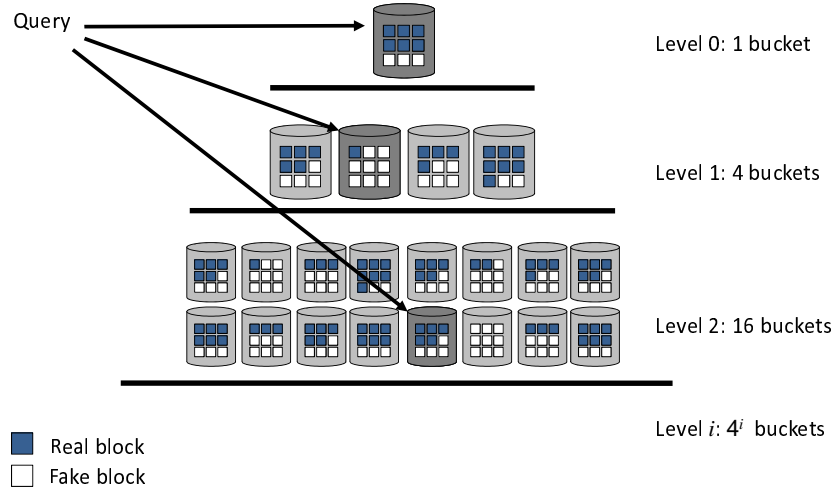


Fig. 1. ORAM: Query Overview. Clients search from the top down, choosing the buckets indicated by a hash function. Since previously accessed items are located in the top level, and buckets are selected randomly on lower levels once an item is found, all accesses appear uniformly random to a polynomial-time observer.

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.

We note that in both *read* and *write* procedures, no block is ever deleted, meaning that the old block values are kept in the database, analogously to the construction of [Goldreich and Ostrovsky 1996].

Level Overflowing. Once level i is full (each 4^i steps), it is emptied into the level below, as illustrated in figure 2. This lower level is completely re-encrypted, and re-ordered according to a new hash function. Thus, accesses to this new iteration of the lower 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. The client

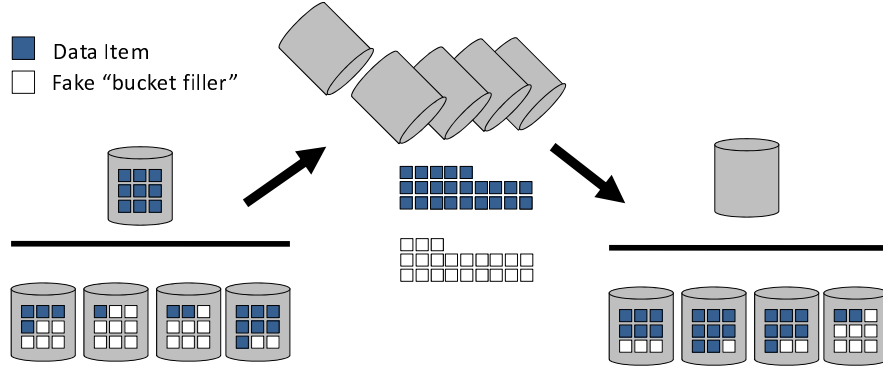


Fig. 2. ORAM: Reshuffling a level into the level below. Once a level is full, the client places all items from that level into the level below, and shuffles obliviously in such a manner that the server can make no correlation between the old locations and the new locations.

decrypts the block, then checks the value of a flag included with the block cleartext, to determine whether it is fake.

Cost. Each query requires a total online cost of $O(\log^2(n))$ for scanning the $\lambda \log n$ -sized bucket on each of the $\log_4 n$ 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 [Goldreich and Ostrovsky 1996]).

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 amount ($\psi(n) = 4c\sqrt{n \log n}$ blocks, where c is a constant) of temporary working memory.

4.1 Additional Client-side Working Memory

Simply adding storage to ORAM in a straightforward manner does not significantly 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 $\log_4 n$ levels, with sizes (number of buckets) 4^i for i from 1 to $\log_4 n$, the blocks belonging to the first $\log_4(\psi(n)) = \log_4(4c\sqrt{n \log n}) = \log_4 4c + \frac{1}{2} \log_4 n + \frac{1}{2} \log_4 \log n$ levels can fit in this storage. This however, would only eliminate slightly more than a constant fraction of the $\log_4 n$ levels, leaving the most expensive levels operating as before.

Second, as indicated in [Goldreich and Ostrovsky 1996], 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 obviously 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 [Ajtai et al. 1983] 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 $\Omega(n \log n)$ [Cormen et al. 2001]. This still results in overall amortized overhead of $\Omega(\log^3 n)$.

Our Approach. 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 $\psi(n)$, with high probability for security parameter c . 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 $\psi(n)$ 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 of size $\lambda \log n$, 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 obviously—without revealing anything about the new permutation to the server.

4.2 Strawman: client with n blocks of working memory

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

Observe that if the client has n blocks of temporary secure storage, it can perform a reorder of level i in only $O(4^i \log n)$ steps, in the following manner. By reading the entire level into the temporary secure storage, throwing out the fake blocks as they are encountered, it can store all 4^i blocks locally. This requires processing $4^i \lambda \log n$ blocks (4^i real ones, and the rest fakes) 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 $O(4^i \log_2(4^i)) = O(i4^i)$. 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 $4^i \lambda \log n$ blocks to the server, for a cost of $O(4^i \log n)$. The total cost of reordering is $O(4^i(i + \log n)) = O(4^i \log n)$.

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 $\log_4 n$ levels total, the amortized computational cost per query of this level reordering approach, across all levels, can therefore be approximated by

$$\sum_{i=1}^{\log_4 n} \frac{O(4^i \log n)}{4^{i-1}} = \sum_{i=1}^{\log_4 n} O(\log n) = O(\log^2 n)$$

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 $\lambda \log n$, the online cost of scanning $\log_4 n$ buckets is $O(\log^2 n)$. Thus the average cost per query, including both online cost and amortized offline cost, 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 $\psi(n)$ blocks of working memory

We will now describe an algorithm for level re-ordering with identical time complexity, but requiring only $\psi(n) = 4c\sqrt{n \log n}$ blocks of local working memory from the client. The client's reordering of level i is divided into Phases (refer to Figure 3). 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), obviously removing the $(\lambda \log n - 1)4^i$ 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 $4^i \lambda \log n$ 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

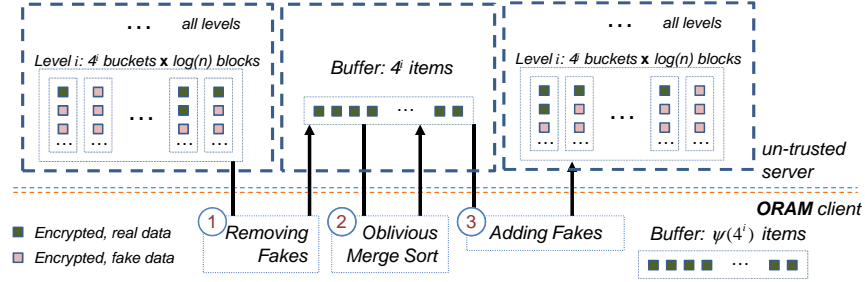


Fig. 3. Solution Overview.

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 computational complexity of this phase is $O(4^i \log n)$. (see Section 4.4)

- (2) **Oblivious Merge Sort.** Obviously 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 computational complexity of this phase is $O(4^i \log n)$. (see Section 4.5)
- (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, obviously adding in the $4^i(\lambda \log n - 1)$ fake blocks necessary to guarantee all buckets have the same size. The computational complexity of this phase is $O(4^i \log n)$. (see Section 4.6)

The above algorithm reorders level i into the new permutation, in time $O(4^i \log n)$. 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 $\psi(n)$ 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 the illustration in Figure 4 and provided pseudocode for Procedure RemoveFakes). 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

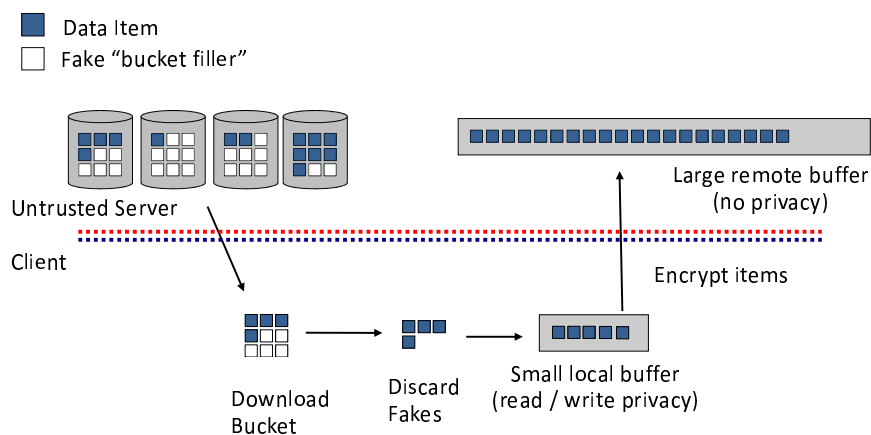


Fig. 4. Reshuffle Phase 1: Remove fakes

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 3 that, with high probability, a queue of size $\psi(n)$ will not overflow or empty out.

```

/*RemoveFakes: Scan all items at level  $i$  on the server, and write them
back excluding the fake ones. Use a local buffer to hide which were
the fake ones. */
 $q \leftarrow$  empty queue stored locally, with room for up to  $\psi(n)$  elements
 $r \leftarrow$  ratio total blocks to real blocks ( $\lambda \log(n)$ )
/*There are  $4^i$  real items and  $(r - 1) * 4^i$  fakes at this level. */
 $y \leftarrow 0$ 
/*Number of blocks output thus far */
for  $x = 1$  to  $r * 4^i + r * \psi(n) / 2$  do
    if  $x < r * 4^i$  then
        /*Read one item on every iteration (until all are read) */
         $t \leftarrow$  decrypt(readNextBlockFromLevel( $i$ ));
        if  $t$  is a real block then
            enqueue( $q$ ,  $t$ )
        end
    end
    if  $y < x / r - \psi(n) / 2$  then
        /*Output one real item every  $r$  interations, starting once the
queue is expected to be about half full. */
         $y \leftarrow y + 1$ 
         $t \leftarrow$  dequeue( $q$ )
        writeNextBlockToRemoteBuffer(encrypt( $t$ ))
    end
end
end
    
```

Procedure: RemoveFakes(i)

This phase requires time linear to the size of the level being read. For level i ,

which contains 4^i buckets of size $\lambda \log n$, 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 (and fakes) is indistinguishable from the case where each real block is assigned a bucket, chosen independently and uniformly at random from the set of all buckets on the actual level.

Before we construct a proof, we consider the notion of a *zero-sum asymmetric random walk*. Given a list of scalars that sum to 0, in the form of sp scalars of value 1 and $s(1-p)$ scalars of value $-\frac{p}{1-p}$, a zero-sum asymmetric random walk consists of the sequence of steps described by a randomly selected ordering of those scalars. We will show that the queue size over an execution of the RemoveFakes algorithm corresponds to a *zero-sum asymmetric random walk*, and moreover, that the overwhelming majority of random orderings result in a walk that remains within $\psi(n)$. We first build probabilistic bounds on the extents of a zero-sum asymmetric random walk.

Probabilistic bounds on a zero-sum asymmetric random walk. Here we are interested in the probability that a walk of length s , starting at 0, and containing $sp+1$ -steps and $s(1-p) - \frac{p}{1-p}$ -steps, exceeds bounds $-c\sqrt{ps} \log s$ or $c\sqrt{ps} \log s$. We show that for any $s > 1$, this probability is negligible in c .

First, we show that we can upper-bound the probability that a walk drawn from a hypergeometric distribution (i.e. containing the fixed number of up-steps and down-steps) exceeds the given bounds by the probability that a walk drawn from the associated binomial distribution exceeds some bounds close to the given ones.

DEFINITION 1. *We say that a sequence $(a_1, \dots, a_s) \in \mathbb{R}^s$ exceeds $u \geq 0$, if for some $k \in \{1, \dots, s\}$, $|\sum_{j=1}^k a_k| > u$.*

To prove the following two statements, the Chernoff bound [Hagerup and Rüb 1990] is used, which we briefly recall.

THEOREM 1 CHERNOFF BOUND. *For i.i.d. random variables $X_j \in \{0, 1\}$ with $\mathbb{E}(X_j) = p$ and $\epsilon > 0$,*

$$\Pr \left[\left| p - \frac{1}{s} \sum_{j=0}^{s-1} X_j \right| \geq \epsilon \right] \leq 2e^{-\frac{\epsilon^2}{4p}s}.$$

LEMMA 1. *Let $Y := (Y_1, \dots, Y_s)$ where Y_j 's are i.i.d. random variables such that $P_{Y_j}(1) = p$ and $P_{Y_j}(-p/(1-p)) = 1-p$. Let us assume that a sequence sampled uniformly from the set of sequences \mathcal{W} of length s , containing $sp+1$ -items and $s(1-p) - p/(1-p)$ -items, exceeds u with probability κ . Then a sequence drawn from P_Y exceeds $u + \delta\sqrt{s}$ with probability at least $\kappa/2$ for $\delta \geq 2\sqrt{\ln(4)p}$.*

PROOF. Note that $sp \in \mathbb{N}$ and $s(1-p) \in \mathbb{N}$. Let \mathcal{U} denote the set of sequences from \mathcal{W} exceeding u . First, notice that all sequences drawn from P_Y and having $s(p+d)$ positive items are equally likely (each of them is just a permutation of another one). Furthermore, any such sequence can be obtained from some sequence in \mathcal{W} , by replacing ds negative items with positive items (if $d > 0$) or by replacing ds positive items by negative items (if $d < 0$). We analyze the first case in detail. The transformation can be done in $\binom{s(1-p)}{ds}$ ways, since a sequence from \mathcal{W} contains

exactly $s(1-p)$ negative items. On the other hand, each sequence having $(p+d)s$ positive items can be obtained from at most $\binom{(p+d)s}{ds}$ sequences from \mathcal{W} . Therefore, replacing ds positive items by negative items in all sequences from \mathcal{U} , maps these sequences into at least $|\mathcal{U}| \binom{s(1-p)}{ds} / \binom{(p+d)s}{ds}$ sequences having $(p+d)s$ positive items. Furthermore, any such sequence exceeds $u - ds(1 + p/(1-p))$. Let a set of these sequences be denoted by \mathcal{U}_d . Let \mathcal{W}_d denote the set of all sequences having $(p+d)s$ positive items. Since

$$|\mathcal{W}_d| = \binom{s}{s(p+d)},$$

we obtain

$$\frac{|\mathcal{W}_d|}{|\mathcal{W}|} = \frac{\binom{s}{s(p+d)}}{\binom{s}{ps}} = \frac{s(1-p) \dots (s(1-p-d) + 1)}{s(p+d) \dots (sp+1)} = \frac{\binom{s(1-p)}{ds}}{\binom{(p+d)s}{ds}}.$$

Consequently,

$$\frac{|\mathcal{W}_d|}{|\mathcal{W}|} = \binom{s(1-p)}{ds} / \binom{(p+d)s}{ds} \leq \frac{|\mathcal{U}_d|}{|\mathcal{U}|},$$

yielding

$$\kappa \leq \frac{|\mathcal{U}|}{|\mathcal{W}|} \leq \frac{|\mathcal{U}_d|}{|\mathcal{W}_d|}.$$

For $d < 0$ we get an analogous result. Therefore, we can write:

$$\begin{aligned} \Pr[Y \text{ exceeds } (u - \delta\sqrt{s})] &\geq \Pr \left[Y \in \bigcup_{j=-\delta\sqrt{s}}^{\delta\sqrt{s}} \mathcal{U}_{j/(s(1+p/(1-p)))} \right] \\ &= \sum_{j=-\delta\sqrt{s}}^{\delta\sqrt{s}} \Pr[Y \in \mathcal{W}_{j/(s(1+p/(1-p)))}] \cdot \Pr[Y \in \mathcal{U}_{j/(s(1+p/(1-p)))} | Y \in \mathcal{W}_{j/(s(1+p/(1-p)))}] \\ &\geq \kappa \sum_{i=-\delta\sqrt{s}}^{\delta\sqrt{s}} \Pr[Y \in \mathcal{W}_{j/(s(1+p/(1-p)))}] = \kappa \left(1 - 2e^{-\frac{(\delta\sqrt{s}/s)^2 s}{4p}} \right) = \kappa \left(1 - 2e^{-\frac{\delta^2}{4p}} \right) \geq \kappa/2, \end{aligned}$$

since according to the assumption for δ , $2e^{-\frac{\delta^2}{4p}} \leq \frac{1}{2}$. To conclude that for each j , $\Pr[Y \in \mathcal{U}_{j/(s(1+p/(1-p)))} | Y \in \mathcal{W}_{j/(s(1+p/(1-p)))}] \geq \kappa$, we use the fact that all walks having the same number of steps up occur with the same probability. Inequality

$$\sum_{j=-\delta\sqrt{s}}^{\delta\sqrt{s}} \Pr[Y \in \mathcal{W}_{j/(s(1+p/(1-p)))}] \geq 1 - 2e^{-\frac{\delta^2}{4p}}$$

follows from the Chernoff bound. \square

Now we are ready to formulate and prove the desired statement.

THEOREM 2. *Let $p < 1/2$. Let W be a random variable distributed uniformly on the set of all walks \mathcal{W} of length s , starting at 0, and containing $sp+1$ -steps and $s(1-p) - p/(1-p)$ steps. For such a walk w , let $f(w) = 1$ if and only if w exceeds*

$c\sqrt{ps \log s}$. Let $F_W := f(W)$. Then for each $s > 1$, $P_{F_W}(1) \leq 2se^{-\frac{c^2 \log s (1-p)^2}{16}}$ which is negligible in c .

PROOF. For a walk y of length s , starting at 0, and containing $+1$ and $-p/(1-p)$ steps, let $f'(y) = 1$ if and only if y exceeds $c\sqrt{ps \log s}/2$. Again, let $Y = (Y_1, \dots, Y_s)$ where Y_j 's are i.i.d. random variables with $P_{Y_j}(1) = p$ and $P_{Y_j}(-p/(1-p)) = 1-p$. Let $F'_Y := f'(Y)$. Lemma 1 then implies that for $c\sqrt{\log s} \geq 2\sqrt{\ln(4)}$,

$$P_{F_W}(1) \leq 2P_{F'_Y}(1),$$

and therefore, it is enough to show that $P_{F'_Y}(1)$ is negligible in c .

To prove the statement, we use the Chernoff bound and the union bound. For $j \in [s]$, let $X_j \in \{0, 1\}$ be i.i.d. random variables with $\mathbb{E}(X_j) = p$. Random variables $Z_j := \frac{1}{1-p}X_j - \frac{p}{1-p}$ then correspond to the steps of a random walk described in the statement. Let $u > 0$. We get $\sum_{i=0}^{s-1} Z_j > u$ if and only if

$$\frac{1}{s} \sum_{j=0}^{s-1} X_j > p + \frac{u(1-p)}{s} = \mathbb{E}(X_j) + \frac{u(1-p)}{s}.$$

Analogously, we get $\sum_{j=0}^{s-1} Z_j < -u$ if and only if

$$\sum_{j=0}^{s-1} X_j < \mathbb{E}(X_j) - \frac{u(1-p)}{s}.$$

According to the Chernoff bound,

$$\Pr \left[\left| p - \frac{1}{s} \sum_{j=1}^{s-1} X_j \right| \geq \frac{u(1-p)}{s} \right] \leq 2e^{-\frac{u^2(1-p)^2}{4ps}}$$

yielding

$$q_s := \Pr \left[\left| \frac{1}{s} \sum_{j=1}^{s-1} Z_j \right| \geq u \right] \leq 2e^{-\frac{u^2(1-p)^2}{4ps}},$$

where q_s denotes the probability that a random walk of length s exceeds either u or $-u$ at the last step. Let p_s denote the probability that a random walk of length s exceeds either u or $-u$ at any step (i.e. exceeds u according to Definition 1). By applying the union bound, we get

$$p_s \leq \sum_{j=0}^{s-1} q_j. \tag{1}$$

Since $g(s) := 2e^{-\frac{u^2(1-p)^2}{4ps}}$ is an increasing function, we can write:

$$p_s \leq \sum_{j=0}^{s-1} q_j \leq 2se^{-\frac{u^2(1-p)^2}{4ps}}. \tag{2}$$

Therefore, p_s is negligible in c if $u \geq c\sqrt{ps \log s}/2$.

It follows that the probability that a walk drawn from P_W exceeds $c\sqrt{ps \log s}$ is also negligible in c . \square

THEOREM 3. *The probability the Remove Fakes queue overflows or empties early is negligible in c , i.e., less than*

$$2(\lambda n \log n) e^{-\frac{c^2 \log(\lambda n \log n) (1 - \frac{1}{\lambda \log n})^2}{16}} \leq 2e^{-c^2 \frac{1}{64}}$$

where $\lambda = \max\{e, \frac{\log 1/p'}{(\log n) \cdot (\log \log n)}\}$ and p' is the acceptable probability of bucket-overflow.

PROOF. First, observe that the real items are positioned according to a uniform random distribution among the fake items. Since a secure hash function builds a uniform random distribution of real items in buckets (each real item is equally and independently likely to end up in any bucket), the fake items are also distributed uniformly randomly; that is, the permutation of real and fake items is chosen with equal probability from all possible permutations.¹

Disregarding for now the first and last $r\psi(4^i)/2$ iterations, which are used to prime the queue at the beginning and empty it at the end, the queue size in the RemoveFakes algorithm can be modeled by a zero-sum asymmetric random walk, according to this distribution of fake and real items. On the iterations where a real item is encountered, the queue size increases by 1 (due to the queue append command). Once every $r = \lambda \log n$ iterations, the queue size decreases by 1.

The queue is easier to model if we assume that instead of decreasing in size by 1 once every r iterations, the decrease is amortized over the r iterations. This idealized queue size differs by at most 1 at any point from the actual queue size. Thus, there are 4^i iterations in which the queue size increases by 1, and in all iterations, the queue size decrease by $1/r$. This is represented by a walk that increases by $1 - 1/r$ in 4^i steps and decreases by $1/r$ in $(r - 1)4^i$ steps. On any particular execution, the ordering of these steps is chosen uniformly randomly from the set of all possible orderings.

Theorem 2 states that zero-sum asymmetric random walk containing sp steps sized $+1$ and $s(1 - p)$ steps sized $-p/(1 - p)$ remains within $\pm c\sqrt{ps \log s}$ with high probability in c . Specifying $s = r4^i$, and $p = 1/r$ proves that that a walk containing 4^i steps sized 1 and $r4^i(r - 1/r) = (r - 1)4^i$ steps sized $-1/r/(1 - 1/r) = -1/(r - 1)$ remains within $\pm c\sqrt{(1/r)(r4^i) \log(r4^i)} = \pm c\sqrt{(4^i) \log(r4^i)}$ with high probability in c .

In order to apply this result, we scale the step size down to $(1 - 1/r)$ on the up-step, multiplying by $1 - 1/r$. This gives us 4^i step-ups of size $1 - 1/r$, and $(r - 1)4^i$ step-downs of size:

$$\frac{1}{r - 1} \left(1 - \frac{1}{r}\right) = \frac{1}{r - 1} \left(\frac{r - 1}{r}\right) = \frac{1}{r}$$

¹The presence of hash collisions (and buckets to resolve these collisions) creates a potential deviation from this idealized uniform distribution. This is resolved if we ensure all items in a bucket are stored in a random permutation. This requirement is unnecessary in practice because our idealized queue size remains within $\pm \lambda \log n$ of the actual queue size, even without the requirement of individual bucket contents being randomly permuted. This is because the buckets are of size $\lambda \log n$.

Thus, a random walk containing 4^i steps sized $1 - 1/r$ and $(r - 1)4^i$ steps sized $1/r$ remains within $\pm(1 - 1/r)c\sqrt{(4^i)\log(r4^i)}$ with high probability in c .

The largest number of real items a level can possibly contain, for a database sized n , is n items. On the bottom level, where $i = \log_4 n$, $4^i = n$ (recall that a level may have from 4^{i-1} to 4^i items during the shuffle; we consider the maximum 4^i items here). On every other level i , $4^i < n$. Thus, since $4^i \leq n$, and with $r = \lambda \log n$, with high probability, the random walk for level i remains under

$$\begin{aligned} \left(1 - \frac{1}{r}\right)c\sqrt{(4^i)\log(r4^i)} &< c\sqrt{n\log(n\lambda\log n)} = c\sqrt{n}\sqrt{\log n + \log\log n + \log\lambda} \\ &< c\sqrt{n}\sqrt{3(\log n)} < 2c\sqrt{n\log n} = \psi(n)/2 \end{aligned}$$

An equivalent symmetric argument holds for the lower bound on the random walk. We have shown that with high probability in c , the walk representing the queue size never exceeds $\pm\psi(n)/2$. The actual queue size must never be negative, of course, and we solve this apparent discrepancy using the first and last $r\psi(n)/2$ steps of the walk. In particular, items are never removed from the queue during the first $r\psi(n)/2$ iterations. This has the effect of increasing the number of items in the queue in relation to the model walk by $(1/r)\psi(n)/2$ each iteration. Thus, past the beginning of the walk, and until the cleanup phase at the end, the actual queue size contains exactly $\psi(n)/2$ items more than the model walk. Thus, since the model walk remains within $\pm\psi(n)/2$, accounting for this difference, the actual queue size remains between 0 and $\psi(n)$.

We just need to show now that the queue size is also within bounds in the beginning and ending phases. In the beginning, observe that an “underflow” is impossible since we are not removing items in this phase. Moreover, since the model walk remains below $\psi(n)/2$, and since our actual queue size is at any point always x more than the model walk, for some $0 \leq x \leq \psi(n)/2$, our actual queue size in the beginning is below $\psi(n)$. Finally, in the ending phase (the last $r\psi(n)/2$ iterations), we are past the end of the walk since we are no longer reading items from the server. The job here is simply to empty out the queue, which contains exactly $\psi(n)$ at the end of the walk, since the model walk ends at 0. Thus, overflow/underflow does not occur here either. \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 s , with $2c\sqrt{s}$ local storage, in $O(s\log_2 s)$ time, without revealing any correlation between the old and new permutations. The algorithm runs recursively on the remote array as described in Procedure ObliviousMergeSort. The recursion depth is $\log_2 s$, and each level of recursion entails a single pass of size $O(s)$ 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 5. Its oblivious nature derives immediately by construction:

```

/*ObliviousMergeSort: Recursively sort the server-stored array  $A$  on
HashLocation( $a$ ) in a manner such that the server learns nothing about
the permutation.  $A$  may be too big to fit entirely in client memory.
*/
if  $A$  is size 1 then
  return  $A$ 
end
 $s \leftarrow$  size of  $A$ 
 $A_1 \leftarrow$  first half of  $A$ 
 $A_2 \leftarrow$  second half of  $A$ 
 $A_1 \leftarrow$  ObliviousMergeSort( $A_1$ )
 $A_2 \leftarrow$  ObliviousMergeSort( $A_2$ )
 $B \leftarrow$  New remote buffer with the same size as  $A$ 
 $q_1 \leftarrow$  empty queue stored locally, to fit up to  $2c\sqrt{s}$  items
 $q_2 \leftarrow$  empty queue stored locally, to fit up to  $2c\sqrt{s}$  items
/*Each queue may contain up to  $2c\sqrt{s}$  items; however  $|q_1| + |q_2| \leq 2c\sqrt{s}$  */
for  $x = 1$  to  $c\sqrt{s}$  do
  enqueue( $q_1$ , decrypt(readNextBlockFrom( $A_1$ )))
  enqueue( $q_2$ , decrypt(readNextBlockFrom( $A_2$ )))
end
/*At this point, each queue will have  $c\sqrt{s}$  blocks */
for  $x = c\sqrt{s} + 1$  to  $s + c\sqrt{s}$  do
  if  $x \leq s$  then
    enqueue( $q_1$ , decrypt(readNextBlockFrom( $A_1$ )))
    enqueue( $q_2$ , decrypt(readNextBlockFrom( $A_2$ )))
  end
  /*Now we've read 2 blocks; time to output 2 blocks */
  for  $i = 1$  to 2 do
     $t_1 \leftarrow$  peek( $q_1$ );
     $t_2 \leftarrow$  peek( $q_2$ );
    if HashLocation( $t_1$ ) > HashLocation( $t_2$ ) then
       $t \leftarrow$  dequeue( $q_1$ )
    else
       $t \leftarrow$  dequeue( $q_2$ )
    end
    writeNextBlockTo( $B$ , encryptWithNewNonce( $t$ ))
  end
end
return  $B$ 

```

Procedure: ObliviousMergeSort(A)

THEOREM 4. *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 supplied pseudocode for ObliviousMergeSort, the `readNextBlockFrom` and `writeNextBlockTo` functions are called in the same pattern every time, depending only on s , not the comparisons made on the output of `HashLocation`.

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 q_1 or q_2). 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 negligible probability, as shown next. \square

Theorem 5 proves that $2c\sqrt{n}$ storage is sufficient for the Oblivious Sort algorithm, with high probability in c . We prove this by showing that all but a negligible portion of the possible permutations encode walks that remain within $\pm c\sqrt{n}$ of the starting location. This is a tighter bound on the required storage bound than what is achieved in Phases 1 and 3; and we note that as an alternative proof, the random walk proof used in Phase 1 can be applied here to achieve a looser bound.

We show now that queues of size $c\sqrt{s}$ are sufficient (with high probability) for random sorting of an array of length s . That is, when selecting elements in a random order from two arrays, the array sizes will likely remain close over time. The queue size can be modeled intuitively as a one-dimensional random walk with s steps, with the additional property, since we select without replacement, that the further we deviate from balance, the more likely each step is to bring us back towards balance.

The queue size at step j in any step of our merge sort algorithm is a probabilistic function $Q(j)$ defined iteratively as follows:

$$\begin{cases} Q(0) = c\sqrt{s/2} \\ Q(j) = Q(j-1) + 1/2 \text{ Pr. } 1/2 - (Q(j-1) - c\sqrt{s/2})/(s-j) \\ Q(j) = Q(j-1) - 1/2 \text{ Pr. } 1/2 + (Q(j-1) - c\sqrt{s/2})/(s-j) \end{cases}$$

On each step, we either dequeue an item from this queue or the other, symmetric queue (not modeled here). If we dequeue an item from this queue, the queue size

will decrease by one; if not, it will remain the same. Additionally, once every two steps we enqueue one item to the queue; our model captures this by enqueueing half an item on every step. This results in stepping $+1/2$ or $-1/2$ (instead of $+0$ or -1) depending on whether we dequeue an item on this step.

We want to show that with high probability, the queue size at any step along the walk will be greater than zero, and less than $2c\sqrt{s/2}$. The step function is chosen to select with equivalent probability one remaining item from either array. Since the arrays are both size $s/2$, when the arrays finally empty, we will have selected an equal number of items from each. Observe that $Q(s) = c\sqrt{s/2}$: as j approaches s , the “pressure” to tend $Q(j)$ to $c\sqrt{s/2}$ increases.

This function is equivalent to pulling our steps out of an urn without replacement, starting with $s/2$ step-ups and $s/2$ step-downs in the urn. The further we deviate from an equivalent number of each step, the more likely it is for the next step to bring the tally closer to equivalent counts.

We will reframe the probabilistic function $Q - c\sqrt{s/2}$ as a random walk of length s , starting and ending at 0, with $d = Q(j) - c\sqrt{s/2}$, according to the following step function, with step size $1/2$.

$$\begin{aligned} P(\text{up}) &= \frac{1}{2} - \frac{d}{s-j} = \frac{(s-j-2d)}{2(s-j)} \\ P(\text{down}) &= \frac{1}{2} + \frac{d}{s-j} = \frac{(s-j+2d)}{2(s-j)} \end{aligned}$$

We define a *biased random walk* to be a sequence of steps chosen according to the step function defined above. We now show that all such biased walks are equally likely. Intuitively, the bias in each step is exactly the amount necessary to choose equally from all the remaining walks that end at 0.

LEMMA 2. *All biased walks are all equally likely*

PROOF. At any position in the walk, 2 steps constituting an up followed by a down can be replaced by a 2 steps constituting a down followed by an up to obtain an equally likely walk. This property falls from our step function, (height d , step j , length s):

$$\begin{aligned} Pr[\text{up}@j, d]Pr[\text{down}@j+1, d+\frac{1}{2}] &= \frac{(s-j-2d)}{2(s-j)} \frac{(s-(j+1)+(2(d+\frac{1}{2})))}{2(s-(j+1))} \\ &= \frac{(s-j-2d)(s-j+2d)}{4(s-j)(s-j-1)} \\ &= \frac{(s-j+2d)}{2(s-j)} \frac{(s-(j+1)-(2(d-\frac{1}{2})))}{2(s-(j+1))} \\ &= Pr[\text{down}@j, d]Pr[\text{up}@j+1, d-\frac{1}{2}] \end{aligned}$$

This shows that at any point in a walk the pair of steps up-down (UD) is as likely to exist as the pair down-up (DU), independent of the rest of the walk. Thus, any such pairs can be interchanged in any walk to find an equally likely walk.

This covers all possible biased random walks, showing they are all equally likely: UUUUDDDD=UUUDUDDD=UUDUDUDD, etc. \square

We define an *unbiased random walk* to be a random walk starting at 0, and defined according to this following step function, with step size 1/2:

$$\begin{cases} Pr[up@j, d] &= 1/2 \\ Pr[down@j, d] &= 1/2 \end{cases}$$

LEMMA 3. *There is a one-to-one correspondence between the biased random walks of length n and the unbiased random walks of length n that end at 0*

PROOF. Every *biased random walk* of length s can be mapped to a unique *unbiased random walk* that has the same sequence of steps, ending at 0. Moreover, every *unbiased random walk* of length s that ends at 0 can be mapped to a unique *biased random walk* that follows the same sequence of steps. \square

According to [Feller 1967], there is a one-to-one correspondence between random walks starting at position $(0, 0)$, reaching (j, d) at some j along the way, and ending at $A = (s, 0)$, with the random walks that start at position $(0, 0)$ and end at $A' = (s, 2d)$. Namely, for every walk that goes through (j, d) and eventually makes it to $(0, 0)$, there is another walk that is equivalent until (j, d) , at which point it begins mirroring the original walk over $y = d$. This provides an easy way to compute the number of walks that go “out of bounds”. This reflection principle is illustrated in Figure 5.

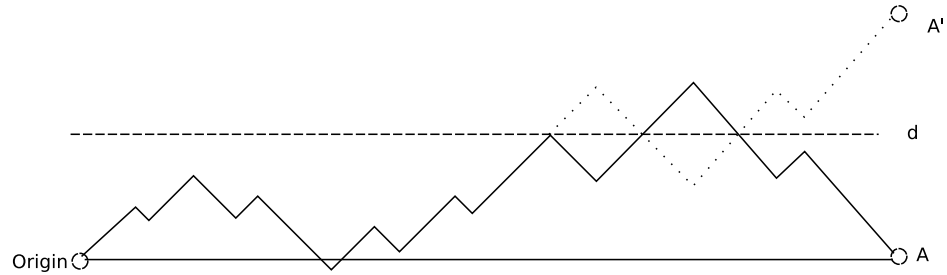


Fig. 5. Feller’s Reflection Principle

LEMMA 4. *There are $\binom{s}{\frac{s}{2}+2d}$ unbiased random walks from 0 to 0 that hit d on the way.*

This lemma is included here for completeness; it may be found in [Feller 1967].

PROOF. A walk of length s from $(0, 0)$ to $(s, 2d)$ consists of $4d$ more step-ups than step-downs (step sizes are still 1/2). Therefore, it must consist of $\frac{s}{2} + 2d$ step-ups and $\frac{s}{2} - 2d$ step-downs. These steps can be in any order; therefore there are a total of $\binom{s}{\frac{s}{2}+2d}$ such walks.

The reflection principle shows that there are hence a total of $\binom{s}{\frac{s}{2}+2d}$ walks from $(0, 0)$ to $(s, 0)$ that hit d somewhere on the way. \square

LEMMA 5. A randomly selected biased random walk hits d with probability $\frac{\binom{s}{s/2+2d}}{\binom{s}{s/2}}$

PROOF. Lemma 4 shows that there are $\binom{s}{s/2+2d}$ walks from $(0, 0)$ to $(s, 0)$ that hit d somewhere on the way. Lemma 3 shows that since each of these unbiased random walks goes from $(0, 0)$ to $(s, 0)$, it constitutes a valid *biased* random walk as well. Therefore, there are $\binom{s}{s/2+2d}$ biased random walks that hit d .

Lemma 2 shows that all of our biased random walks are equally likely. Since there are $\binom{s}{s/2}$ biased random walks total, a randomly chosen biased random walk hits d with probability

$$\frac{\binom{s}{s/2+2d}}{\binom{s}{s/2}}$$

□

Our queue size function is a biased random walk from $(0, 0)$ to $(0, s)$ (with height 0 on the walk corresponding to size $c\sqrt{s/2}$ on the queue). The queue overflows, if at any point, the walk is above $c\sqrt{s/2}$ (and “underflows” if below $-c\sqrt{s/2}$).

Lemma 5 shows that the chance a uniformly randomly selected biased walk hits d is $\binom{s}{s/2+2d}/\binom{s}{s/2} = \binom{s}{s/2-2d}/\binom{s}{s/2}$. By showing this quantity is negligible in c when $d = c\sqrt{s/2}$, we now show that with high probability the queue will remain within bounds.

We upper-bound the value of $\frac{\binom{s}{s/2-2d}}{\binom{s}{s/2}}$ as follows: We have that

$$\begin{aligned} \frac{\binom{s}{s/2-2d}}{\binom{s}{s/2}} &= \frac{\frac{s!}{(s/2-2d)!(s/2+2d)!}}{\frac{s!}{(s/2)!(s/2)!}} \\ &= \frac{(s/2)(s/2-1)\dots(s/2-2d+1)}{(s/2+2d)(s/2+2d-1)\dots(s/2+1)}. \end{aligned}$$

Since the function $f(x) = \frac{x}{x+a}$ is increasing for $a > 0$, we conclude that

$$\begin{aligned} \frac{\binom{s}{s/2-2d}}{\binom{s}{s/2}} &= \frac{(s/2)(s/2-1)\dots(s/2-2d+1)}{(s/2+2d)(s/2+2d-1)\dots(s/2+1)} \\ &\leq \left(\frac{s/2}{s/2+2d}\right)^{2d}. \end{aligned}$$

For $d = c\sqrt{s/2}$ we get:

$$\begin{aligned} \frac{\binom{s}{s/2-2c\sqrt{s/2}}}{\binom{s}{s/2}} &\leq \left(\frac{s/2}{s/2+2c\sqrt{s/2}}\right)^{2c\sqrt{s/2}} \\ &= \left(1 - \frac{1}{\sqrt{s/2}(1/(2c) + 1/\sqrt{s/2})}\right)^{2c\sqrt{s/2}}. \end{aligned}$$

For $s \geq 2$, $1/\sqrt{s/2} \leq 1$ and thus we obtain:

$$\begin{aligned} \frac{\binom{s}{s/2-2c\sqrt{s/2}}}{\binom{s}{s/2}} &\leq \left(1 - \frac{1}{\sqrt{s/2}(1/(2c)+1)}\right)^{2c\sqrt{s/2}} \\ &= \left(\left(1 - \frac{1}{\sqrt{s/2}(1/(2c)+1)}\right)^{\sqrt{s/2}(1/(2c)+1)}\right)^{2c/(1/(2c)+1)}. \end{aligned} \quad (3)$$

We substitute $m := \sqrt{s/2}(1/(2c)+1)$ to find that:

$$\left(1 - \frac{1}{\sqrt{s/2}(1/(2c)+1)}\right)^{\sqrt{s/2}(1/(2c)+1)} = (1 - 1/m)^m \leq e^{-1}.$$

From (3) we now conclude that

$$\frac{\binom{s}{s/2-2c\sqrt{s/2}}}{\binom{s}{s/2}} \leq e^{-2c/(1/(2c)+1)}.$$

For $c \geq 1/2$, we therefore obtain

$$\frac{\binom{s}{s/2-2c\sqrt{s/2}}}{\binom{s}{s/2}} \leq e^{-c}.$$

It remains to mention that in fact, for sufficiently large s , the value of $1/\sqrt{s/2}$ can be arbitrarily small, which implies that for large s , the term $e^{-2c/(1/(2c)+1)}$ can be replaced by $e^{-2c(1/(2c)+1/\sqrt{s/2})}$ which can be made smaller than $e^{-2c(1/(2c)+\delta)}$ for any $\delta > 0$ by choosing s sufficiently large. Therefore, the upper bound on the probability that a walk exceeds the allowed bounds can in fact be made close to e^{-4c^2} . It follows that to get the probability of a random walk of length s exceeding bounds $\pm c\sqrt{s/2}$ smaller than 2^{-100} , it is sufficient to set $s \geq 300$ and $c \geq 6$.

THEOREM 5. *With high probability, Oblivious Merge Sort queues never overflow or empty early.*

PROOF. We have already established that the queue size is a biased random walk of length s , offset by the constant $c\sqrt{s}$. Therefore, the chance that a queue hits $2c\sqrt{s}$ is $\leq e^{-c}$. Symmetrically, the chance that a queue hits 0 is also $\leq e^{-c}$, so the chance that a queue goes out of bounds during one instance of the merge sort phase is $\leq 2e^{-c}$. This is negligible for a security parameter c , which is independent of s .

For sorting of level i , the number of steps in the walk is $s \leq 4^i \leq n$. Thus, with high probability in c the walk remains within $\pm c\sqrt{n}$, and $2c\sqrt{n}$ storage suffices. \square

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.

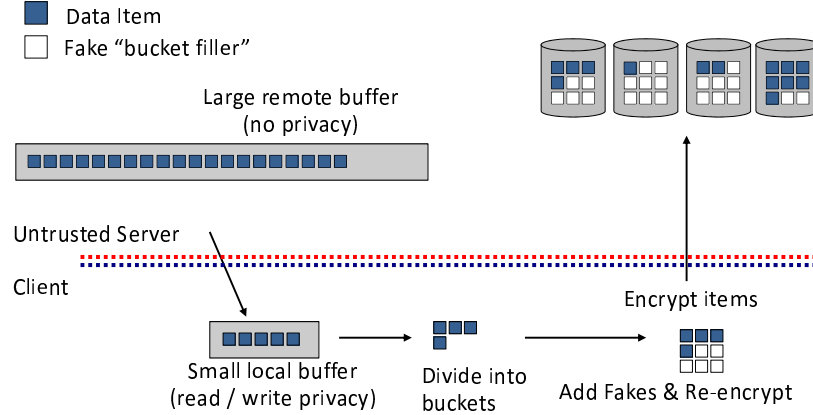


Fig. 6. Reshuffle Phase 1: Add fakes

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 (as in ORAM; see Section 3). At the same time, fake blocks are added to make all buckets mutually indistinguishable. This is the exact inverse of Phase 1. Pseudocode is provided for Procedure `AddFakesToLevel`; an illustration is provided in Figure 6.

For correctness we must also show here that the buckets of size $\lambda \log n$ will not overflow. It is easy to see that if s blocks are randomly thrown into s buckets, the fullest bucket has no more than t blocks with probability at most $\frac{s}{t!}$. This follows from the fact that each bucket gets at least t blocks with probability at most $\binom{s}{t} \frac{1}{s^t} \leq \frac{1}{t!}$ and the union bound. For $t = \lambda \log s$, the probability that each bucket gets at most t blocks is therefore at least $1 - s^{-\lambda \cdot (\log \log s + \log(\lambda/e))}$. If we fix the probability of bucket-overflow to be at most p' , then for a database of size n (which is the size of the lowest pyramid level), $\lambda \log n$ -sized buckets are sufficient for $\lambda = \max\{e, \frac{\log 1/p'}{(\log n) \cdot (\log \log n)}\}$. Note that for all practical purposes, λ can be considered a constant once p' is fixed. For example, for a reasonable $p' = 2^{-100}$, if $n > 2^{14}$ (more than 16384 blocks) $\lambda = e$.

When blocks are placed into the buckets on higher levels, having less than n buckets, the probability of bucket-overflow is upper-bounded by the probability that buckets on the lowest level overflow. In the event that an overflow ever does occur, the original ORAM [Goldreich and Ostrovsky 1996] prescribes a level re-order abort and restart; the expected time complexity of that algorithm is not affected. However, notice that by conditioning on the fact that no bucket ever overflows, we differentiate the distribution of bucket assignments from the truly uniform one. Since the ORAM algorithm outlined in Section 3 prescribes to randomize the pyramid traversal once the desired block is found, this can potentially become a privacy leak. Hence, we need to ensure that throughout the whole program execution, the need for restarting a level re-order happens with sufficiently low probability. If the probability of bucket-overflow for one level-reorder step is at most p' , then the

```

/*AddFakesToLevel: Scan all items in array  $A$  on the server, which are
sorted by the bucket they belong in, and write them to level  $i$ , adding
fakes to make equal-sized buckets. Use a local buffer to hide which
are the fake ones. */
 $q \leftarrow$  empty queue stored locally, with room for up to  $\psi(n)$  elements
 $r \leftarrow$  ratio total blocks to real blocks ( $\lambda \log(n)$ )
 $y \leftarrow 0$ 
/*Number of blocks read so far */
 $b \leftarrow 0$ 
/*Current bucket number */
 $q' \leftarrow$  empty queue stored locally, with room for  $r$  elements
for  $x = 1$  to  $r * 4^i + r * \psi(n) / 2$  do
  if  $y < x/r$  and  $y < 4^i$  then
    /*Input one real item every  $r$  iterations. */
     $y \leftarrow y + 1$ 
    enqueue( $q$ , decrypt(readNextBlock( $A$ )));
  end
  if  $x > r * \psi(n) / 2$  then
    /*Add one item to the bucket every iteration, real if there is
one waiting for this bucket, fake otherwise (starting once the
queue is about half full) */
     $t \leftarrow$  peek( $q$ )
    if BucketNumber( $t$ ) =  $b$  then
      enqueue( $q'$ , encryptWithNewNonce(dequeue( $q$ ),  $t$ ))
    else
      enqueue( $q'$ , encryptWithNewNonce(fake))
    end
  if  $b < x/r - r * \psi(n) / 2$  then
    /*Output one bucket every  $r$  iterations. */
    writeBucketToLevel( $q', i$ )
     $b \leftarrow b + 1$ 
    empty( $q'$ )
  end
end
end
end

```

Procedure: AddFakesToLevel(A, i)

union bound yields the probability of level re-order restart throughout the whole execution of an ORAM program (query sequence) to be at most $p' \cdot k$, for k being the number of level re-order steps. Since the i -th level is re-ordered every 4^i steps, where $i > 0$, for a query sequence of length K , there are $k \leq \sum_{i=1}^{\infty} \frac{K}{4^i} = \frac{K}{3}$ level re-order steps. Hence, the probability of bucket-overflow in any step for a query sequence of length K is at most $Kp'/3$.

As in Phase 1, we employ a local buffer of size $\psi(n)$ 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. 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 up to 4^i buckets of size $\lambda \log n$, the running time is $O(4^i \log n)$.

THEOREM 6. *With high probability, the Add Fakes algorithm queue never overflows or empties early.*

PROOF. The result is a random distribution of fake and real items. This distribution encodes a zero-sum asymmetric random walk, as in the Remove Fakes algorithm. Every iteration adds one item to a bucket. On iterations in which there is a real item to add (on average once every r iterations), the queue size decreases by 1. When we read from A , once every r iterations, the queue size increases by 1. As in the analysis of the RemoveFakes algorithm, we instead consider the queue size to increase by $1/r$ every iteration, giving us 4^i step-downs of size $1 - 1/r$ and $(r - 1)4^i$ step-ups of size $1/r$. The remainder of this proof is an exact parallel of Theorem 3, with the step sizes inverted. \square

THEOREM 7. Correctness. *After Phase 3, all blocks will be in the correct bucket (determined by the secure hash function).*

PROOF. 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. \square

THEOREM 8. 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. Theorem 4 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. \square

5. DISCUSSION AND EXTENSIONS

Handling Duplicates. While for simplicity, we did not discuss this aspect in the paper, the following scenario can ensue and requires handling. Read blocks are added back to the top of the ORAM, resulting in potentially conflicting versions that need to be reconciled. To prevent this, we perform an extra step in the online query phase: scanning a bucket entails downloading the bucket to client storage, removing the item we were looking for and replacing it with a fake item, re-encrypting, and

writing it back. This results in doubling the network traffic in the online query phase: every bucket read is accompanied by a write. This is reflected in Figure 8.

Goldreich and Ostrovsky’s ORAM handles this scenario by reconciling multiple versions of an item in the reshuffle phase; as an alternative to the above online-phase mechanism we could adopt a similar approach. This will improve the overhead of the online phase in exchange for extra work in the reshuffle. To achieve this, we add an extra pass during the reshuffle to remove duplicates after Phase 2 (Oblivious Merge Sort), then repeat Phase 2. Recall that after the Phase 2, the blocks are sorted by the buckets they will end up in. This single pass reads blocks, re-encrypts them, replaces them with fakes if they are duplicates, and writes them back, ensuring there are no duplicates during the second iteration of Phase 2. This step can be performed without affecting the asymptotic complexities since the communication, and computational, and storage requirements, are less than the requirements of the other phases of the shuffle.

Asymptotic Considerations. To maintain indistinguishability between reads and inserts, the database size increases on every access. Of course, this property may not be desirable for all users; if not, it is simple to consolidate the duplicates periodically, at the expense of privacy.

This growth affects computation of the asymptotic complexity. Indeed, for simplicity reasons, the above complexity is an overestimation of the actual cost seen in an implementation. This is so because earlier queries operate over a smaller n . Nevertheless, in computing the complexity, we consider the current, larger n . Overall, this should not significantly diverge from the actual seen amortized cost since the cost of recent queries dominates. To see why this is the case, consider that every $4n$ accesses a new level is created, resulting in the size of the database effectively quadrupling in size. Thus, for any considered period ending in the creation of a new level, we are overestimating at most for a quarter of the queries.

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 requires a client to be able to download from a public server, meaning the client does not have access to prearranged secret keys. By implementing the access pattern privacy on a server-side tamper-proof trusted processing component such as a Secure Co-processor, we can also achieve PIR. The secure CPU maintains the encrypted database, and never leaks any of the encryption keys to the server or clients. Clients who wish to retrieve an item privately then interact with the main data through the SCPU.

We will consider predicted performance in Section 6. Taking the IBM 4764 as an example, figure 10 shows that when we implement PIR on the secure CPU, the bottleneck is no longer the network bandwidth, but the en/decryption times. Additionally of concern is the limited SCPU storage. The 64MB of RAM available on a 4764 can support a 10GB database with the probability of overflowing the queue space of less than 2^{-194} , derived from the bounds established in Theorem 3.

Finally, for a fixed size of the database, larger blocks require more storage on the client side to achieve the same privacy guarantees. For a database of size m , with $n = m/b$ blocks of size b , $O(b\sqrt{n \log n}) = O(\sqrt{mb(\log m - \log b)})$ storage is needed. Hence, the amount of storage scales proportionally to \sqrt{b} .

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 640 MB instead of just 64 MB; this is enough allow the provider to support 1 TB databases.

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 SCPUs may see a much higher idle time.

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

The PIR protocol introduced in [Wang et al. 2006] offers an amortized complexity of $O(n/k)$ for database size n and secure storage size k . For $k = O(\sqrt{n \log n})$, this yields an overhead of $O(\sqrt{n/\log n})$ 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.

Implementation: Required Storage Memory.

As can be seen in the pseudocode for Procedure AddFakesToLevel, in the actual implementation, we use slightly more than $\psi(n)$ storage. In particular, we need room for a small amount of additional storage, e.g., to store temporary data. In the provided pseudocode this amounts to one more bucket (sized $\lambda \log n$).

6. PERFORMANCE

	Server	Client
RAM	4GB	1GB
processor		2Ghz
disk seek time	5ms	
sustained disk read/write	50 MB/s	
Link bandwidth		10 MB/s
Link round trip time		50ms
En/Decryption		100 MB/s (Based on processor speed, using AES [Lipmaa])
Outsourced data set size	1 TB, in 1000-byte blocks; $n = 10^9$	

Fig. 7. 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 7. Further, Figure 10 illustrates multiple

	Formula	Sample
Network latency	$RTT_{link} \cdot \log_4(n)$	750ms
Disk seek	$Latency_{seek} \cdot \log_4(n) \cdot 2$	150ms
Network transfer	$\log_4(n) \cdot \lambda \log(n) \cdot 2 \cdot \text{blocksize} / \text{Throughput}_{link}$	168ms
Client en/decryption	$\log_4(n) \cdot \lambda \log(n) \cdot 2 \cdot \text{blocksize} / \text{Throughput}_{crypto}$	17ms
Server disk transfer time	$\log_4(n) \cdot \lambda \log(n) \cdot 2 \cdot \text{blocksize} / \text{Throughput}_{disk}$	34ms

Fig. 8. **Online** cost per query, resulting from scanning a bucket at each level.

such data points. We set the bucket size to be $\lambda \log n = \epsilon \log 10^9$. Using the formula from Section 4.6, we obtain the probability of bucket-overflow to be at most 2^{-100} , which is low enough in practice to assume that no bucket ever overflows.

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 8 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.

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 $4^i \lambda \log n$, 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 copying back $4^i \lambda \log n$ 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 9 shows the resulting formulas. If all of these costs are incurred sequentially, we have an amortized response rate of approximately $1.5s/query$ offline plus $1.5s/query$ online, for $3s/query$.

	Formula	Sample	
Network latency.	n/a	< 1ms	Level reordering is not interactive, so idling can be avoided here. The Phase 1 and 3 scans account for its bulk. Seek time will be hidden by disk transfer during reordering. This load can be split among several disks.
Network transfer	$\frac{\log_4(n) \cdot \lambda \log(n) \cdot 10 \cdot \text{blocksize}}{\text{Throughput}_{link}}$	842ms	
Disk seek latency	n/a	< 1ms	
Disk transfer	$\frac{\log_4(n) \cdot \lambda \log(n) \cdot 10 \cdot \text{blocksize}}{\text{Throughput}_{disk}}$	168ms	
Client processing	$\frac{\log_4(n) \cdot \lambda \log(n) \cdot 10 \cdot \text{blocksize}}{\text{Throughput}_{crypto}}$	84ms	

Fig. 9. Amortized **offline** cost per query.

The bottleneck when determining the parallel query throughput is the network throughput, at $842ms/query$. Thus, when making simultaneous use of multiple resources by running queries in parallel, this results in a query throughput of over a query per second.

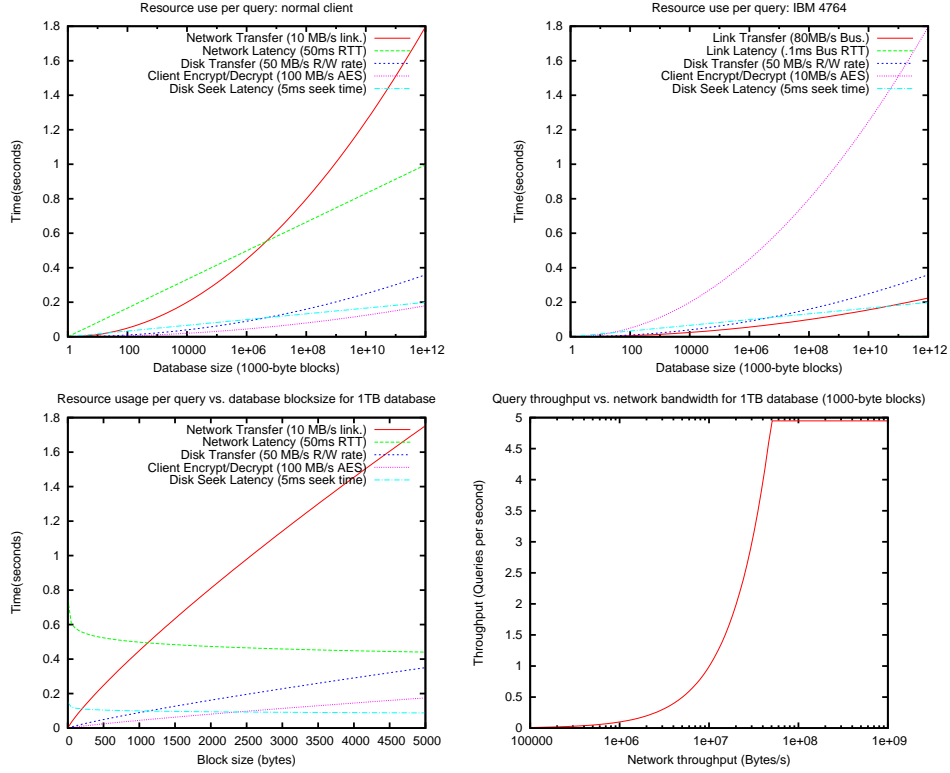


Fig. 10. Amortized use of resources per query (a) for a normal client and (b) using a IBM 4764 SCPU as a client. (c) resource usage as the the database block size is varied, and (d) estimated query throughput as the the network link capacity is varied. Both (c) and (d) assume a fixed database size of 1TB.

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 requiring $4^i \log(n) \log^2(4^i \log n)$ block transfers, for a total of $10 \cdot 4^i \cdot \log(n) \cdot \log^2(4^i \log n) \cdot 2^{10} / 10MB/sec$. 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 $\sum_{i=1}^{15} 10 \cdot 4 \cdot 15 \cdot \log^2(15 \cdot 4^i) \cdot 2^{10} B = 614KB \sum_{i=1}^{15} (\log 15 + \log 4^i)^2 \approx 3.680GB$. Over the sample 10MB/s link this is a 368 sec/query amortized transfer time, almost three orders of magnitude slower.

7. CONCLUSIONS

We introduced a (first) practical oblivious RAM 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 $O(\log n \log \log n)$ per query.

REFERENCES

- AJTAI, M., KOMLOS, J., AND SZEMEREDI, E. 1983. An $O(n \log n)$ sorting network. In *Proceedings of the 25th ACM Symposium on Theory of Computing*. 1–9.
- ASONOV, D. 2004. *Querying Databases Privately: A New Approach to Private Information Retrieval*. Springer Verlag.
- CHOR, B., GOLDREICH, O., KUSHILEVITZ, E., AND SUDAN, M. 1995. Private information retrieval. In *IEEE Symposium on Foundations of Computer Science*. 41–50.
- CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill.
- FELLER, W. 1967. *An Introduction to Probability Theory and its Applications*. Vol. 1. Wiley.
- GARTNER, INC. 1999. Server Storage and RAID Worldwide. Tech. rep., Gartner Group/Dataquest. www.gartner.com.
- GASARCH, W. 2004. A survey on private information retrieval.
- GASARCH, W. 2010. A WebPage on Private Information Retrieval. Online at <http://www.cs.umd.edu/~gasarch/pir/pir.html>.
- GOLDBERG, I. 2007. Improving the Robustness of Private Information Retrieval. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*.
- GOLDREICH, O. 2001. *Foundations of Cryptography*. Cambridge University Press.
- GOLDREICH, O. AND OSTROVSKY, R. May 1996. Software protection and simulation on Oblivious RAM. *Journal of the ACM* 43, Issue 3, 431–473.
- HAGERUP, T. AND RÜB, C. 1990. A guided tour of chernoff bounds. *Inf. Process. Lett.* 33, 6, 305–308.
- HILD, M. AND MITCHELL, J. 2004. Free Email: Google, MSN Hotmail and Yahoo! (A). *SSRN eLibrary*.
- IBM CORP. 2008. IBM 4764 Model 001 Specification Sheet. Online at http://www-03.ibm.com/security/encryptocards/pdfs/4764-001_PCIX_Data_Shee%t.pdf.
- ILIEV, A. AND SMITH, S. 2004. 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*. 201–216.
- LIPMAA, H. Aes ciphers: speed. Online at <http://research.cyber.ee/~lipmaa/research/aes/rijndael.html>.
- SASSAMAN, L., COHEN, B., AND MATHEWSON, N. 2005. The pynchon gate: a secure method of pseudonymous mail retrieval. In *WPES*. 1–9.
- SCRIBNER, C. 2007. Comment and casenote: Subpoena to Google Inc. in *ACLU v. Gonzales: "big brother" is watching your internet searches through government subpoenas*. *University of Cincinnati Law Review* 75, 1273.
- SION, R. AND CARBUNAR, B. 2007. On the Practicality of Private Information Retrieval. In *Proceedings of the Network and Distributed Systems Security Symposium*. Stony Brook Network Security and Applied Cryptography Lab Tech Report 2006-06.
- WANG, S., DING, X., DENG, R. H., AND BAO, F. 2006. Private information retrieval using trusted hardware. In *Proceedings of the European Symposium on Research in Computer Security ESORICS*. 49–64.
- WILLIAMS, P., SION, R., AND CARBUNAR, B. 2008. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *ACM Conference on Computer and Communications Security*. 139–148.
- YANG, Y., DING, X., DENG, R. H., AND BAO, F. 2008. An efficient PIR construction using trusted hardware. In *Information Security*. Lecture Notes in Computer Science, vol. 5222. Springer Berlin / Heidelberg, 64–79.