Single Round Access Privacy on Outsourced Storage

Peter Williams and Radu Sion
Network Security and Applied Cryptography Lab
Stony Brook University, Stony Brook, NY, USA
{petertw,sion}@cs.stonybrook.edu

ABSTRACT

We present SR-ORAM,[1] the first single-round-trip polylogarithmic time Oblivious RAM that requires only logarithmic client storage. Taking only a single round trip to perform a query, SR-ORAM has an online communication/computation cost of $O(\log n \log \log n)$, and an offline, overall amortized per-query communication cost of $O(\log^2 n \log \log n)$, requiring under 2 round trips. The client folds an entire interactive sequence of Oblivious RAM requests into a single query object that the server can unlock incrementally, to satisfy a query without learning its result. This results in an Oblivious RAM secure against an *actively malicious* adversary, with unprecedented speeds in accessing large data sets over high-latency links. We show this to be the most efficient storage-free-client Oblivious RAM to date for today's Internet-scale network latencies.

Categories and Subject Descriptors

D.0 [**Software**]: General; E.3 [**Data Encryption**]

Keywords

Access Privacy, Cloud Computing, Oblivious RAM

1. INTRODUCTION

Oblivious RAM (ORAM) allows a client to read and write data hosted by an untrusted party, while hiding both the data and the *access pattern* from this untrusted host. Access pattern privacy is a critical component of data privacy. Without access pattern privacy, the act of reading and writing remote data leaks potentially essential information about the data itself, making it impossible to achieve full data confidentiality. Since the introduction of the first Oblivious RAM in [6], approaches to increase query throughput have been relentlessly sought. Nevertheless, and despite the wide range of potential applications, practical Oblivious RAM constructions have remained elusive until very recently.

One of the most significant challenges to providing practical ORAM is that these interactive protocols require a large number of client-server round trips, resulting in large, often impractical, online query latencies. For example, the construction presented by Goodrich et al. [8] requires $\log_2 n$ round trips, translating to an online cost alone of over 1200-1500ms per query on a 1 terabyte database (e.g., for 10KB blocks), assuming a network link with a latency of just 50ms.

This paper provides a simple and direct solution to the challenge: SR-ORAM, a single-round-trip ORAM. Querying requires a single message to be sent from the client to the server and thus incurs a single round-trip (for a total online cost of 50ms in the example above). Moreover, SR-ORAM does not greatly affect the offline, amortized cost.

The basic idea behind SR-ORAM is to fold the interactive queries into a single non-interactive request without sacrificing privacy. The client constructs a set of values (a "query object") that allows the server to selectively decrypt pieces, depending on new values obtained during its traversal of the database. Each component of the query object unlocks only a specific single new component—which allows server database traversal progress while preventing it from learning anything about the overall success of the query.

Our construction is based on the Bloom filter ORAM of [19], since it lends itself conveniently to use of a non-interactive query object and provides defenses against actively malicious adversaries (not only curious). We also make use of the randomized shell sort defined in [9], since it allows the more stringent client storage requirements of SR-ORAM (when compared to [19]).

Other ORAMs with constant numbers of round trips exist; Section 3 reviews recent solutions. However, SR-ORAM is the first to provide a constant-round-trip *polylogarithmic time* construction assuming only *logarithmic client storage*.

2. MODEL

A capacity-constrained *client* desires to outsource storage to an untrusted party (the *server*). The client has enough local non-volatile storage to manage keys and certificates, plus enough volatile RAM to run the ORAM client software (logarithmic in the size of the outsourced data). Moreover, since the client reads and writes sensitive data, it needs to hide both the data content and access pattern. Thus, the client needs low-latency, private access to this remote disk.

Data is accessed in "blocks", a term used to denote a fixed-size record. "Block" is used instead of "word" to convey tar-

[1]A preliminary version of this paper appeared in the ACNS Industrial Track [18].

get applications broader than memory access (file system and database outsourcing, in particular, are lucrative targets). Block IDs are arbitrary bit sequences sized $O(\log n)$.

Participants

Communication between the user and the ORAM Client is secured, e.g., with access controls on inter-process communication if they are on the same machine, or with SSL otherwise. Communication between the ORAM Client and ORAM Server is also secured, e.g., with a transport-layer protocol such as SSL.

ORAM Client: the trusted party providing the following (self-explanatory) interface to the user: read(id): val; write(id, val). The Client-Server protocol details are implementation specific (and typically optimized to the instance to minimize network traffic and the number of round trips).

The client keeps track of two values between queries: its secret key and the current access count. From this, the current level keys, and the reshuffle count of each level, can be derived.

ORAM Server: the untrusted party providing the storage backend, filling requests from the instance.

Security Definitions

We will defend against curious, potentially malicious (not constrained to follow the protocol) polynomially bounded adversaries. We operate in the random oracle model[2]. The actively malicious defense is inherited from the underlying ORAM, described by Williams et al. [19].

For simplicity, timing attacks are not discussed here. Defenses include the introduction of client-side delays to uniformize query times—which can be done without affecting overall protocol complexity. Additionally, SR-ORAM assumes semantically secure symmetric encryption primitives and secure hash functions.

Notation

Throughout the paper, n refers to the database size, in blocks. The client secret key is sk. The number of times a given level has been shuffled (i.e. reconstructed) is called the "generation," and is abbreviated as gen. Oblivious RAM is ORAM; Bloom Filter is BF. Key size and hash function output size are both assumed to be c_0; c_1 is the Bloom filter security parameter.

To represent computational and communication costs in a comparable manner, complexities are represented in words, not bits. It is assumed that each word can hold an entire identifier, e.g., $O(\log n)$ bits.

3. BACKGROUND

We start with an review of ORAM, and in particular, Bloom-filter-based ORAMs. We next review the highly interactive Bloom-filter-based ORAM [19], which provides a convenient construction to build SR-ORAM from. Finally, we look at recent approaches to reduce the round trip cost.

3.1 ORAM Overview

Oblivious RAM[6] provides access pattern privacy to *a single client* (or software process) accessing a remote database (or RAM). The amortized communication and computational complexities of the construction by Goldreich and Ostrovsky [6] are $O(\log^3 n)$ for a database sized n. This construction requires only logarithmic storage at the client.

We now begin a review of their logarithmic construction, on which our own is modeled. The server-hosted database is a set of n semantically secure encrypted blocks (with a *secret key* held by the client). Supported operations are read(id), and write(id, $newvalue$). The data is organized into $\log_2(n)$ *levels*, as a pyramid. Level i consists of up to 2^i blocks; each block is assigned to one of the 2^i buckets at this level as determined by a hash function.[3] Due to hash collisions each bucket may contain from 0 to $O(\log n)$ blocks. [4]

ORAM Reads. To obtain the value of block id, a 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 the secret key and a different nonce (so it looks different to the server) 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 is 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 henceforth be completely independent of any previous accesses. Each level overflows once the level above it has been emptied twice. The resulting 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 (e.g., [1] or [9]) is used to re-order the blocks thusly.

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*

[2]For an ORAM that does not assume access to a random oracle, we refer the reader to the construction and analysis by Damgård et al. [5]

[3]$\log_4(n)$ levels sized 4^i in the original, but for simplicity we use a branch factor of 2.

[4]This was originally specified as $\log n$ blocks, with a non-negligible probability of bucket overflow, in which case a new hash function is tried. It was later shown [13] that this results in an information leak.

blocks. Recall that since every block is encrypted with a semantically secure encryption function, the adversary cannot distinguish between fake and real blocks.

3.2 Bloom filters

Bloom filters [3] offer a compact representation of a set of data items. They allow for relatively fast set inclusion tests. Bloom filters are *one-way*, in that, the "contained" set items cannot be enumerated easily (unless they are drawn from a finite, small space). Succinctly, a Bloom filter can be viewed as a string of b bits, initially all set to 0. To *insert* a certain element x, the filter sets to 1 the bit values at index positions $H_1(x), H_2(x), \ldots, H_k(x)$, where $H_1, H_2, \ldots, H_k$ are a set of k crypto-hashes. Testing set inclusion for a value y is done by checking that the bits for *all* bit positions $H_1(y), H_2(y), \ldots, H_k(y)$ are set.

By construction, Bloom filters feature a controllable rate of false positives r for set inclusion tests—this rate depends on the input data set size z, the size of the filter b and the number of cryptographic hash functions k deployed in its construction: $r = \left(1 - (1 - 1/b)^{kz}\right)^k$.

As will be seen below, the SR-ORAM Bloom filters are constrained by two important considerations. First, we need to minimize k, since this determines directly the number of disk reads 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, since a false positive reveals lookup failure to the server.

Encrypted Bloom Filters. The idea behind remotely-stored encrypted Bloom filters is to store their bit representation encrypted while still allowing client-driven Bloom filter lookups. This can be achieved, e.g., by storing the Bloom filters as bit strings XORed with client-side PRNG-driven key strings, or with individual bits stored and encrypted separately with semantic security (at the expense of additional storage).

As will be shown, in SR-ORAM, instead of storing an encrypted bit for each position of the Bloom filter, we store part of a decryption *key*. Since the server cannot distinguish between the keys for bit-values of *1* and the keys for bit-values of *0*, we retain the property that the server does not learn the success of the Bloom filter lookup.

3.3 Bloom filter-based ORAMs

The main contribution of [19] is the separation of level membership testing from item storage. Instead of checking for an item at a given level by reading the entire relevant bucket of $O(\log n)$-blocks, an encrypted Bloom filter is queried first. This indicates to the client which of two potential items (the real, if there, or a specific fake, otherwise) to retrieve. This saves a factor of $O(\log n)$ server storage, as there is now only a single fake item per real item, instead of $O(\log n)$. These reduced storage requirements simultaneously speed up level construction and querying.

More specifically, item location is encoded via a Bloom filter; any given item has membership in one of $\log_2 n$ Bloom filters, corresponding to one for each level. The level corresponding to the Bloom filter that contains this item is the level where the item must be retrieved from. Bloom filters are queried from the top down; maintaining access privacy requires that any given lookup be performed only once on any given Bloom filter. Once an item is found to be at a particular level, it is retrieved by its label, which is a one-way keyed hash of the item id, level number, and the number of times this level has been rebuilt. It is then copied up to the top, so the request will be satisfied at a higher level next time. Fake lookups are performed on those levels where the item does not exist. These retrieve a previously stored fake item identified by a one-way keyed hash of the level number, the number of times this level has been rebuilt, and the current total access count. Random Bloom filter lookups are performed below where the item is found.

This is an interactive process requiring $\log_2 n$ round trips: the client needs to know the success of a lookup at a given level before it can start the query at the next level. Figure 1 illustrates this process of querying.

It has been shown [14] that the security analysis of [19] is incomplete, suggesting larger Bloom filters are needed to obtain negligible false positive rates. They also recommend a different selection in the tradeoff between Bloom filter size (affecting server storage and shuffle cost), and the number of hash functions chosen (affecting online cost). This adds a factor of $\log \log n$ to the Bloom filter construction cost. We apply these insights in the choices of Bloom filter parameters (number of hash functions k, and size in bits) and in the performance analysis (Section 7) of SR-ORAM.

We also note that [19] assumes a significant amount of temporary client storage necessary in the reshuffle step. This assumption is not suitable for our model. Instead, following the example of others[14, 7], SR-ORAM uses an oblivious randomized shell sort [9] to support the level reshuffle and construct Bloom filters obliviously without client storage. This reduction in client storage requirements comes with performance penalties, as will be discussed later.

3.4 Other constant-round-trip ORAMs

Other recent approaches provide ways around the penalty of highly interactive protocols, at the cost of additional hardware or overwhelming requirements of client storage. The main issue in constructing a single round trip ORAM is that a request for an item depends on how recently an item was accessed. Maintaining this information at the client requires storage at least linear in the size of the outsourced database. Moreover, retrieving this information privately from the server is almost as difficult as providing ORAM. [5]

Secure Hardware. Secure hardware such as the IBM 4764 [11] can be placed server-side, using remote attestation to retain security guarantees for clients [2]. Secure hardware is typically an order of magnitude more expensive than standard processors. Due to heat dissipation difficulties it is typically also an order of magnitude slower. Moreover, the necessity of physical security to provide any guarantees makes such solutions vulnerable to a different class of attacks. Several authors have examined the use of secure hardware in the access privacy scenario [15, 12].

Constant-round-trip protocols using client storage. [16] maintains item location information at the client. Although at the outset, $n \log_2 n$ bits of client storage seems like a big assumption, the authors argue this is reasonable in some situations, since the block size is typically larger than $\log n$. They show that in practice, the local required client storage in practice is only a small fraction of the total database size. The recursive construction, using a second

[5]With the difference that this recursive ORAM only requires storing $O(\log \log n)$ bits per item, which is enough location information about the item to build the query.

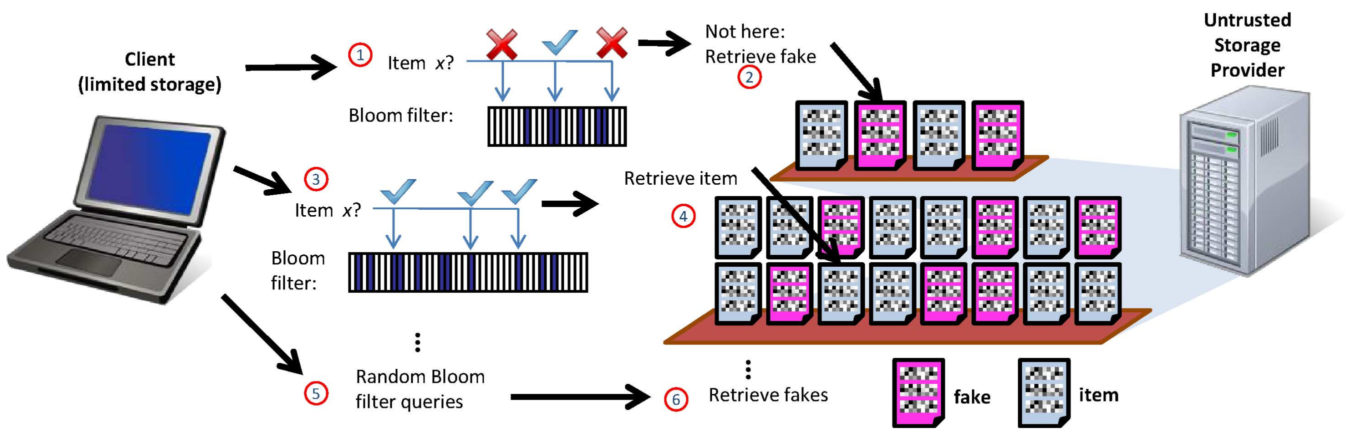


Figure 1: Interactive Bloom filter querying. Both the lower level Bloom filter lookups and item lookups are dependent on the Bloom filter results of the levels above.

ORAM to store this level membership information, however, is interactive. SR-ORAM requires only $O(\log_2 n)$ bits of client storage (Section 5).

More recently, Goodrich et al. [10] introduced a constant-round-trip protocol assuming $n^{1/c}$ client storage. The number of round trips depends on the amount of client storage.

The non-interactive cache-based ORAM presented in 2006 by Wang et al. [17] relies on s client storage to provide an amortized overhead of $O(n/s)$. The idea is to add previously unseen items to a cache, which gets shuffled back into the remote database when it fills. The high client storage requirements (and poor storage/performance tradeoff) make it unsuitable for our model. This idea is revisited under different assumptions by Boneh et al. [4], with security formalization, but still requiring client storage.

A large number of *interactive* ORAM solutions have been proposed. A great review is provided by Kushilevitz et al. [13]. A full review should also include recent interactive *de-amortized* ORAMs such as the construction by Goodrich et al. [8]. These resolve another drawback of many ORAMs (SR-ORAM included), the disparity between average-case and worst-case query cost.

4. A FIRST PASS

This strawman construction modifies a Bloom-filter-based ORAM presented by Williams et al. [19]. It has the structure, but not yet the performance, of the SR-ORAM construction. As detailed in Section 3.3, that Bloom filter ORAM uses encrypted Bloom filters to store level membership of items. To seek an item, the querying client must request a known fake item from each level, except from the level containing this item: the item is requested here instead. Which level the item is at depends only on how recently this item was last accessed. Since the client does not have storage to keep track of that, it checks the Bloom filters one at a time to learn if the item is at each level.

As the main principle of level-based ORAMs requires *each item be sought only once per level instance*, it is unsafe to query the Bloom filters past the level where this item is present. This explains why the checks must be interactive: once the item is found at level i, further accesses at the levels below ($i+1$ through $\log_2 n$) entail only random Bloom filter queries corresponding to fake item requests. Moving the found item to the top of the pyramid guarantees that later, it will be sought and found elsewhere, since it moves back down to other levels only by riding a wave of level reshuffles.

We now turn this, safely, into a non-interactive process. Observe that in an interactive ORAM, if the client is requesting a recently accessed item j that happens to be in level 2, the access sequence will proceed as follows. This example is also illustrated in Figure 1. We use j to denote the item identifier, and *sk* for the secret key, and *gen* to represent the current generation of that level (a function of the total, global number of accesses, *accesscount*).

1. The client checks the level 1 Bloom filter for the item: reading the positions generated by:
   Hash(sk | $level$=1 | gen | j)

2. Upon seeing Encrypt(*0*) at one or more of those positions in the Bloom filter, the client learns the item is not at level 1. So it asks for a fake item instead, that is labeled as:
   Hash(sk | $level$=1 | gen | "fake" | $accesscount$)

3. The client now checks the level 2 Bloom filter for the item: reading the positions indicated by:
   Hash(sk | $level$=2 | gen | j)

4. Seeing Encrypt(*1*) at every position, the client learns the item is at this level. This makes it safe to request the item here; the client requests the block labeled
   Hash(sk | $level$=2 | gen | "item" | j)

5. Having found the item, to maintain appearances and not reveal this fact to the server, the client continues to issue random Bloom filter lookups at each level i below. At each level it requests the fake blocks labeled
   Hash(sk | $level$=i | gen | "fake" | $accesscount$)

Note that there are only $\log_2 n$ possible such access sequences, based on which level the item is found at (Figure 2). Each path starts with a real query. Active queries continue until an item is found, at which point only fake queries are issued from there on down. This limited number of possible sequences makes non-interactive querying possible.

Since we have a finite number of these paths, our goal is to follow one of these paths non-interactively, not knowing

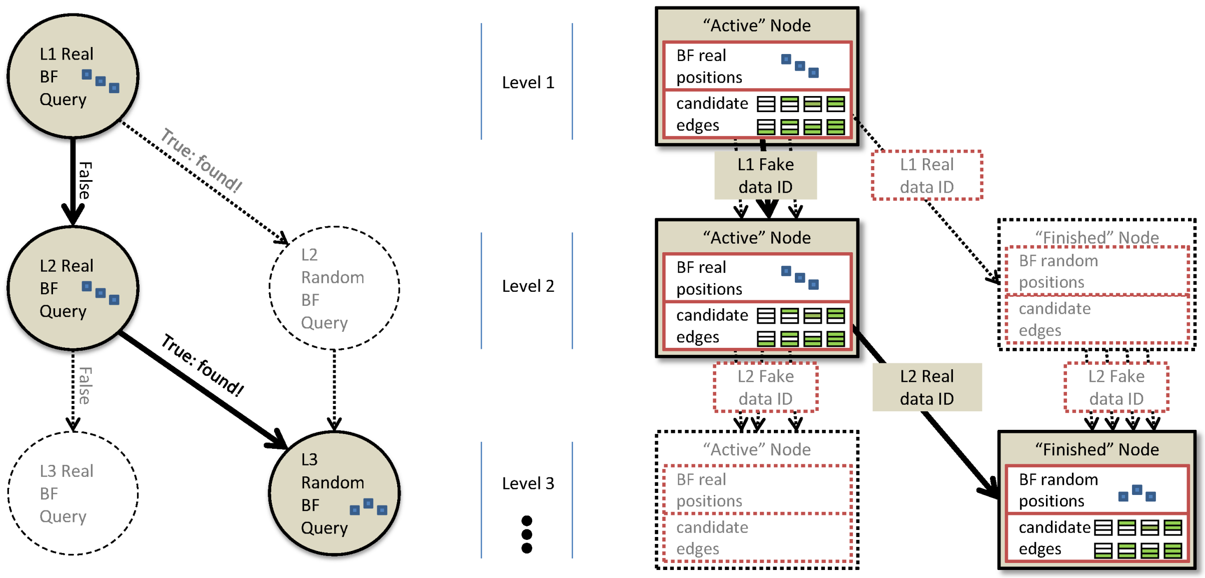


Figure 2: Left: potential query paths in unmodified BF-based ORAM. The client does not know at the time of querying which of the $\log_2 n$ possible paths will be taken; it depends on where the data is ultimately found. As in the example used in Figure 1, the level 1 BF lookup returns false, indicating a fake should be retrieved from level 1. The level 2 BF lookup returns true, indicating the item should be retrieved from level 2. A fake BF query is run for level 3 since the item has already been found. Right: the query object in SR-ORAM. The server learns the edges corresponding to exactly one path. The server will be able to decrypt one such edge at each level, revealing the data ID, to retrieve and include in the response to the client, and decrypting a node of the query object in the level below.

ahead of time which level the item is at (and thus which of the $\log_2 n$ paths will be followed).

To achieve this, we propose to have the Bloom filter results themselves be used in unlocking one of the two possible edges leading to the next query. A successful lookup will unlock the edge leading to a "finished" set, under which only fake queries will follow. Conversely, failure must unlock the edge continuing down the "active" search set. Once in the "finished" set, it is impossible to return back to the "active" set. Most importantly, the server must not gain any ability at identifying which path it is currently on.

One strawman idea, exponential in the number of Bloom filter hashes k, is to *make each bit in the Bloom filter a piece of a decryption key* unlocking an edge to the next node. For each level, the client prepares 2^k results, corresponding to each possible state of the Bloom filter. The Bloom filter keys are generated deterministically by the client using a cryptographic hash, so that the client can efficiently keep track of them with only logarithmic storage. That is, a bit set to *1* at position *pos* in the Bloom filter is represented by $T_{pos} = \text{Hash}(sk \mid pos \mid \text{level} \mid \text{gen} \mid 1)$, and a bit set to *0* by $F_{pos} = \text{Hash}(sk \mid pos \mid \text{level} \mid \text{gen} \mid 0)$. The server learns only one of the two (*never both*).

A Bloom filter lookup involves k bit positions (k is the number of underlying Bloom filter hash functions). For each new level it traverses, the server needs to know the k associated Bloom filter bit positions to retrieve, constituting this level's query. *For the first level, these are provided by the client.* For each successive level, the server will get this information by incrementally decrypting portions of a client-provided "query object" data structure.

Illustrated in Figure 2 (right), the "query object" is composed of $\log_2 n$ levels and is traversed by the server top-down synchronized with the traditional ORAM traversal. The query object allows the server to progress in its database traversal without learning anything.

Each level in the query object (with the exception of the root), contains two nodes: a "finished" node and an "active" node. Each node contains the k positions defining the current level Bloom filter query. The nodes also contain a "keying" set of 2^k elements.[6]

After performing the Bloom filter lookup, the server will be able to decrypt one of these elements (only). Once decrypted, this element contains a key to decrypt one of the query object's next level two nodes; it also contains the identifier for a current level item to return to the client. To prevent leaks, the server will be asked to return one item for each level, since we do not want to reveal when and where we found the sought-after real item.

In effect this tells the server where to look next in the query object—i.e., which of the query object's next level two nodes ("finished" or "active") to proceed with. This guides the server obliviously through either the "finished" or the "active" set, as follows:

- If the current level contains the sought-after item, the server's work is in fact done. However, the server cannot be made aware of this. Hence, it is made to continue its traversal down the ORAM database, via a sequence of fake queries. The "finished" node of the next query object level allows the server to do just

[6]After encryption, these elements are sent in a random order to prevent the server from learning any information.

that, by providing the traversal information down the "active" set.

- If, however, the current level *does not* contain the sought-after item, the server must be enabled to further query "real" data in its traversal down the ORAM database—it will thus receive access to "active" node of the next query object level.

To prevent the server from decrypting more than one element from a node's "keying" set, a special encryption setup is deployed. Each of the 2^k elements of the "keying" set is encrypted with a special query object element key (QOEK), only one of which the server will be able to reconstruct correctly after its Bloom filter query.

More specifically, for a Bloom filter lookup resulting in k bit representations (i.e., bit_i is the representation of the bit at position i – either T_i or F_i [7]), the QOEK is defined as QOEK $= \text{Hash}(bit_1 \mid bit_2 \mid bit_3 \mid ... \mid bit_k)$.

The encryption setup of the "keying" set ensures that this key decrypts exactly one of its elements. The element corresponding to a Bloom filter "hit" (the sought-after element was found at this level, i.e., all the underlying Bloom filter bits are set to *1*) leads down the "finished" set, i.e., the element that QOEK decrypts now, leads down the "finished" set in the query object's next level.

5. EFFICIENT CONSTRUCTION

We now present an efficient construction, using $O(\log n)$ client storage, $O(\log n \log \log n)$ per-query online message size, and $O(\log^2 n \log \log n)$ amortized communication, but still only $O(1)$ round trips. We reduce the size of the query object of Section 4 from $2^k \log n$ to just $k \log n$.

The main insight is to allow compression of the 2^k decryption key possibilities into only $k+1$ possibilities. This is achieved by representing the Bloom filter bits and their combination in a commutative format. By allowing the decryption key pieces stored in the Bloom filter (described in the previous section) to be added together, rather than concatenated, the client only has to account for $k+1$ different outcomes at each level.

To this end, we start by first establishing a secret level-instance-specific token $v = \text{Hash}(sk \mid level \mid gen)$, not known to the server. In the Bloom filter, a bit set to *1* at position pos is represented as $T_{pos} = \text{Hash}(sk \mid level \mid gen \mid pos)$; a bit set to *0* is represented as $F_{pos} = T_{pos} + v \mod 2^{c_0}$ (Figure 3), where c_0 is a security parameter. In the following we operate in $\mathbb{Z}_{2^{c_0}}$, and assume that $\text{Hash}(\cdot)$ outputs in $\mathbb{Z}_{2^{c_0}}$.

Now, the query object encryption key (QOEK) is generated by combining (adding) values using modular arithmetic, instead of concatenation as in the strawman solution. This allows the client to only account for $k+1$ possibilities, each key corresponding to the number of times v might show up among the selected Bloom filter positions.

The server sums together the values found in the Bloom filter, and performs a hash, yielding, e.g., $\text{Hash}(bit_1 + bit_2 \ldots + bit_k \mod 2^{c_0})$ as the QOEK. The commutativity of modular addition means each permutation of bits set to *0* and bits set to *1* in a given Bloom filter yields the same key. A successful Bloom filter lookup occurs in the case that the QOEK is $\text{Hash}(T_{pos_0} + T_{pos_1} + ... + T_{pos_k} \mod 2^{c_0})$, which unlocks the edge from the "active" to the "finished" set.

Further, each of the k values from $\text{Hash}(T_{pos_0} + T_{pos_1} + ... + T_{pos_k} + v \mod 2^{c_0})$ through $\text{Hash}(T_{pos_0} + T_{pos_1} + ... + T_{pos_k} + kv \mod 2^{c_0})$—the result of a failed Bloom filter lookup; the server does not know they are failures—unlocks an edge in the query object that continues through the current ("finished" or "active") set (Figure 2). Figure 4 provides a summary of the query object format.

Let us now consider an example from the perspective of the server. Say the client sends a query object for an item x. This query object contains a single Level 1 node in cleartext, and two nodes for every level below. The Level 1 active node tells it to query positions $L1pos_1 \ldots L1pos_k$. The server retrieves the values stored in the Bloom filter at these locations, adds them modulus 2^{c0}, and applies the one-way hash function. This yields a decryption key. The server now tries this key on all k encrypted values included in the Level 1 node. It finds one that successfully decrypts, revealing a data ID to retrieve from Level 1, as well as the decryption key for one of the two Level 2 nodes. The server appends the retrieved Level 1 data item to its result, then decrypts the Level 2 node that it has the key for.

The server now repeats for the Level 2 node. It finds a list of Bloom filter positions, which it again retrieves, adds modulus 2^{c0}, and hashes, yielding a decryption key which it tries on the encrypted values included in the Level 2 node. Again, only one will decrypt. The server never learns which are the active or finished nodes; it simply sends back $\log_2 n$ data values to the client, of which one will be a real item, and the others fake items.

5.1 Obliviously building the Bloom filter and levels

This section describes how to construct the Bloom filter without revealing to the server which Bloom filter positions correspond to which items. Further, it shows how to obliviously scramble the items as a level is constructed. Both processes are mostly non-interactive (only a constant small number of round trips).

In the Bloom filter construction, the key privacy requirement is that the server is unable to learn ahead of time any correlation between Bloom filter positions and data items. Constructing a new level in a BF-based ORAM requires first randomly and obliviously permuting all the items, renaming them according to the new level hash function, and introducing a fake item for each (again, obliviously, and using the appropriate hash-function defined name). Constructing the Bloom filter requires first scanning the new set of items, building an encrypted list of positions that will be need to set, and then obliviously rearranging this encrypted list into the appropriately-sized segments of the resulting encrypted Bloom filter. The result is a pristine new level, generated using a deterministic read and write pattern (independent of the contents or history).

Further, note that we differ from previous work in that we store decryption keys in the Bloom filter, instead of single encrypted bits. Recall that these components are computed on the client by a secure keyed hash of the position, and added to the value v if this position is intended to represent a bit value of *0*.

The other main difference from existing work is that [19] assumes a significant amount of temporary client storage,

[7]Recall that bit_i does not reveal to the server anything about the actual underlying Bloom filter bit since the server does not know the hash key sk.

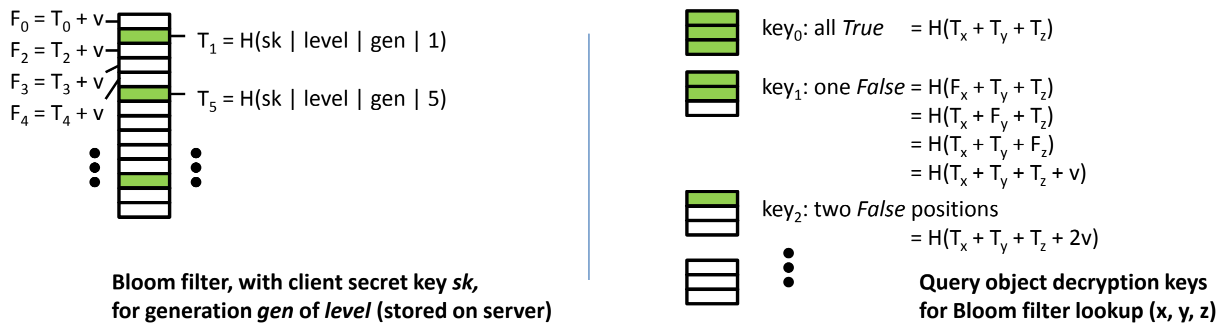

$F_0 = T_0 + v$
$F_2 = T_2 + v$
$F_3 = T_3 + v$
$F_4 = T_4 + v$
T_1 = H(sk | level | gen | 1)
T_5 = H(sk | level | gen | 5)
Bloom filter, with client secret key *sk*, for generation *gen* of *level* (stored on server)
key_0: all *True* = $H(T_x + T_y + T_z)$
key_1: one *False* = $H(F_x + T_y + T_z)$ = $H(T_x + F_y + T_z)$ = $H(T_x + T_y + F_z)$ = $H(T_x + T_y + T_z + v)$
key_2: two *False* positions = $H(T_x + T_y + T_z + 2v)$
Query object decryption keys for Bloom filter lookup (x, y, z)

Figure 3: Left: Bloom filter format in full construction. A bit set to *1* in the Bloom filter is represented by a hash of the position and current level key; a bit set to *0* is represented by the same value, plus the secret value v. Right: decryption keys used with the query object. The server obtains the decryption key for a given query stage by hashing together the specified Bloom filter results. Since there are k hash functions used in a Bloom filter check, the client includes in the query object an edge corresponding to each of the $k+1$ possible keys.

which is not suitable in our model. To avoid this requirement we propose to use two passes of an $O(n \log_2 n)$ oblivious randomized shell sort from [9].

This shell sort will be applied to a list produced by the client as follows. The client starts by producing a list of (encrypted) positions that need to be set in the Bloom filter. The client will then also add a number of "segment delimiters" to this list. These delimiters will aid later. One delimiter is issued per segment (e.g., 32 adjacent positions) in the Bloom filter. These delimiters will later provide an excuse to output something for positions that are *not* to be set, to prevent the server from learning which bits are set.

The client then performs a first sorting pass, outputting the list sorted by positions, with the delimiters interspersed (delimiters include information that allows their sorting). This sorted list is then scanned, and, for each of its non-delimiter elements, a fake 32 bit value is issued. For each encountered segment delimiter however, a 32 bit (encrypted) segment of the Bloom filter is output. This segment's bits are set correctly according to the recently seen (since the last segment's delimiter encounter) encrypted set positions.

This simple mechanism prevents the server from learning how many bits are set in each segment. To complete the process, a second oblivious sort pass then moves the fake 32 bit values to the end of this new list, where they can be safely removed by the server.

Finally, the encrypted bit-storing Bloom filter needs to be converted into a key-storing Bloom filter in one final step: in a single non-oblivious pass, we read each bit and output either T_{pos} for *1*s and F_{pos} for *0*s (where *pos* is the current bit's position in the Bloom filter). Note this multiplies the size of the remotely stored object by the key size in bits.

We do not need to modify the level construction from [19], except in replacing their storage-accelerated merge sort with the storage-free randomized shell sort.

Non-interactivity of sort. It is worth noting that the shell sort, like the storage-accelerated merge sort, can be implemented in a small, constant number of round trips. Each step in both sorting algorithms requires reading two items from the server, and writing them back, possibly swapped (which is an interactive process). However, the item request pattern of both sorting algorithms is known ahead of time to the server. This allows it to send data to the client ahead of time, without waiting for the request from the client (alternately, the client can issue requests for future items far in advance of the time they will be used).

This process is trivial in the merge sort: the access pattern consists of simultaneous scans of two (or sometimes up to $\sqrt{n}$) arrays; the server streams these to the client. This process of non-interactive streaming of sort data to the client is not as trivial in the randomized shell sort. Once the random seed is chosen, however, both the client and server know in advance the order the items will be requested in. The server can run a simulation of this sort, for example, to know which items the client needs to read next, and avoid waiting for network round-trips throughout the construction of the level.

The end result in both scenarios is a sort process whose cost is almost completely independent of the network latency. For any sort, regardless of the size, the client first sends an initialization message. Then, the server sends a long stream of all the item contents to the client, as the client simultaneously writes a long stream of the permuted item contents back to the server.

6. SECURITY

SR-ORAM directly inherits the privacy properties of the server-side ORAM database traversal, as well as the integrity defenses from the base ORAM construction in [19]. We must now establish the privacy of the query object construction, as well as the new Bloom filter construction.

We establish privacy of the query object construction in Theorem 1. The server learns only one set of Bloom filter positions and one item label to retrieve at each level for a given query. In other words, the server sees only what it would see in an equivalent, interactive instantiation.

LEMMA 1. *The server gains no non-negligible advantage at guessing v = Hash (sk | level | gen) from observing (i) the Bloom filter contents or (ii) the hashes included in the query object.*

PROOF. (sketch) For each position *pos* in the Bloom filter, the server sees a single value X that is either a random number T_{pos}, or $T_{pos} + v \mod 2^{c_0}$. If T_{pos} and v are both chosen randomly from $\mathbb{Z}_{2^{c_0}}$, then seeing the entire set of values gives the server no knowledge of v.

- Level 1:
  - L1 active node, in cleartext:
    - Level 1 Bloom filter lookup index positions $L1pos_1 \dots L1pos_k$ (integer values)
    - the client computes, *but does not send*, these $k+1$ keys:
      - $key_{L1,\text{ success}} = \text{Hash}(T_{L1pos_1} + T_{L1pos_2} + \dots + T_{L1pos_k})$
      - $key_{L1,1} = \text{Hash}(T_{L1pos_1} + T_{L1pos_2} + \dots + T_{L1pos_k} + v)$

        ...
      - $key_{L1,k} = \text{Hash}(T_{L1pos_1} + T_{L1pos_2} + \dots + T_{L1pos_k} + kv)$
    - $k+1$ encrypted values, included in a random order:
      - $E_{key_{L1,\text{ success}}}$ (L1 **real** data ID, and key2F: the key for the L2 finished node)
      - $E_{key_{L1,1}}$ (L1 fake data ID and key2A: the key for the L2 active node)

        ...
      - $E_{key_{L1,k}}$ (L1 fake data ID and key2A: the key for the L2 active node)
- Level 2: Both the L2 active and the L2 finished nodes, in a random order:
  - L2 active node, encrypted with randomly generated key2A:
    - Level 2 Bloom filter lookup index positions $L2pos_1 \dots L2pos_k$
    - the client computes, *but does not send*, these $k+1$ keys:
      - $key_{L2,\text{success}} = \text{Hash}(T_{L2pos_1} + T_{L2pos_2} + \dots + T_{L2pos_k})$
      - $key_{L2,1} = \text{Hash}(T_{L2pos_1} + T_{L2pos_2} + \dots + T_{L2pos_k} + v)$

        ...
      - $key_{L2,k} = \text{Hash}(T_{L2pos_1} + T_{L2pos_2} + \dots + T_{L2pos_k} + kv)$
    - $k+1$ encrypted values, included in a random order:
      - $E_{key_{L2,\text{success}}}$ (L2 **real** data ID, and key3F)
      - $E_{key_{L2,1}}$ (L2 fake data ID and key3A)

        ...
      - $E_{key_{L2,k}}$ (L2 fake data ID and key3A)
  - L2 finished node, encrypted with randomly generated key2F:
    - k random Bloom filter lookup index positions for Level 2 $L2pos_1 \dots L2pos_k$
    - the client computes, *but does not send*, these $k+1$ keys:
      - $key_{L2,0} = \text{Hash}(T_{L2pos_1} + T_{L2pos_2} + \dots + T_{L2pos_k})$
      - $key_{L2,1} = \text{Hash}(T_{L2pos_1} + T_{L2pos_2} + \dots + T_{L2pos_k} + v)$

        ...
      - $key_{L2,k} = \text{Hash}(T_{L2pos_1} + T_{L2pos_2} + \dots + T_{L2pos_k} + kv)$
    - $k+1$ encrypted values, included in a random order:
      - $E_{key_{L2,0}}$ (L2 fake data ID and key3F)
      - $E_{key_{L2,1}}$ (L2 fake data ID and key3F)

        ...
      - $E_{key_{L2,k}}$ (L2 fake data ID and key3F)
- Both the L3 active node (encrypted with key3A) and the L3 finished node (encrypted with key3F), in a random order.
- And so forth, for each of the $\log_2 n$ levels.

Figure 4: Query Object Format. For each level, the query object is composed of two possible nodes (with the exception of the top which only has one node). This set constitutes a total of $2\log n$ nodes (containing associated Bloom filter requests), and $2k\log n$ edges. Of these nodes, the server will eventually be able to unlock $\log n$. Each of the unlocked ones provides k edges, of which the server will be able to unlock exactly one. These edges contain the decryption key for a single node at the next level, as well as the data item ID to retrieve. All addition is done modulus 2^{c_0}.

The only other values observed by the server that are linked to v are the outputs of the one-way hash functions, included in the query objects. Seeing the hash outputs in the query object provides no advantage at determining the inputs, beyond using dictionary attacks which are infeasible for a computationally bounded adversary (in the security parameter c_0), since the inputs all include the random v, which is c_0 bits long. $\square$

There is one subtlety here: if the server has some external knowledge about how recently a given query was last issued, it can determine that the set of corresponding Bloom filter positions at that level are all *1*s (and thus, that the values observed at some positions are in fact $T_{pos} + v \mod 2^{c_0}$, rather than T_{pos}). However, knowing this still does not give the server any knowledge of v: all possible values of T_{pos} and v are still equally possible and likely.

LEMMA 2. *For a constant $u > 1$, a Bloom filter using k hashes, with the size-to-contained items ratio of $k \times u$, has a false positive rate bounded by u^{-k}.*

PROOF. Suppose a Bloom filter contains z items and has a size-to-items ratio of $k \times u$. Thus, the size of the Bloom filter in bits is $b = z \times k \times u$. The false positive rate is (Section 3.2)

$$r = \left(1 - \left(1 - \frac{1}{b}\right)^{kz}\right)^k \le \left(\frac{zk}{b}\right)^k = u^{-k} \quad \square$$

LEMMA 3. *For a security parameter c_1, a constant $u > 1$, and a number of $\log n$ lookups on Bloom filters with size-to-contained items ratios of $k \times u$, where $k = c_1 + \log_u \log n$ hashes, the overall false positive rate is negligible in c_1.*

PROOF. From Lemma 2 we know that the false positive rate for any one lookup is $r \le u^{-k}$. Taking the union bound over $\log n$ queries, the probability of failure is bounded by $r \log n \le u^{-c_1}$ which is negligible in c_1. $\square$

THEOREM 1. *The server can only unlock one path down the query object, and all paths appear identical to the server.*

PROOF. (sketch) Each of the $2\log_2 n$ nodes in the query object, as illustrated in Figure 2, provides simply a list of Bloom filter indices, and a set of encrypted values unlocking edges to one of the two next nodes. The negligible Bloom filter false positive rate, established by the base Bloom filter ORAM, guarantees that the set of indices will be unique for every query. Moreover, these indices, chosen by a keyed secure hash, are indistinguishable from random.

Each edge contains an item label which is again uniquely requested, only up to once per level. The label is also determined by the keyed secure hash. Since the contents of any single edge for a given query is indistinguishable from random, and since these edges are included in the query object in a random order, unlocking the edge provides no information to the server about the path.

The contents of the Bloom filter provide the key and determine which one edge the server can unlock. Since, as shown in Lemma 1, the server has no advantage at determining the blinding value v, the server has no advantage at guessing the alternate value at any position of the Bloom filter, and is limited to a dictionary attack against the ciphertexts of the other edges. Thus, the server can only unlock the edge corresponding to the Bloom filter contents. $\square$

Theorem 2. *The server learns nothing from the Bloom filter construction.*

Proof. (sketch) All Bloom filter construction processes, for any particular level, appear to the server to be independent of which bits are set and which items are represented. The server sees only the number of bits in the Bloom filter, which is known beforehand. □

Theorem 3. *An honest but curious adversary gains no non-negligible advantage at guessing the client access pattern by observing the sequence of requests.*

Proof. (sketch) Because of Theorem 1, the adversary learns nothing it would not learn by observing a standard, interactive, Bloom filter-ORAM. We defer to the security claims of previous ORAMs (e.g. [19]) and Lemma 3 which shows the probability of Bloom filter failure to be negligible in the considered security parameter. □

Theorem 4. *An actively malicious adversary has no advantage over the honest but curious adversary at violating query privacy.*

Proof. (sketch) In the underlying Bloom filter ORAM [19], the client detects server protocol deviation, preventing the server from learning anything new from issuing incorrect responses. The non-interactive construction creates a slight difference in the Bloom filter authenticity check: the Bloom filter correctness check is now implicit. That is, the server can only unlock the key if the stored value is the one placed by the client, whereas in previous Bloom filter ORAMs, the client had to test the authenticity of the stored Bloom filter bits before it was safe to continue the query. □

7. ANALYSIS

Following from the construction in Section 5, the query object size is $O(\log n \log\log n)$. It consists of $2\log n - 1$ nodes, each of which queries $k = \log\log n$ Bloom filter positions. This is transmitted to the server, which performs $k \log n$ decryption attempts (of which $\log n$ are successful) before sending $\log n$ blocks back to the client. This yields the online cost of $O(\log n \log\log n)$.

The amortized offline cost per query considers the time required to build each level. A level sized z is built twice every z queries. Shuffling these items using a randomized shell sort costs $O(z \log z)$. Since, as shown in Lemma 3, the Bloom filter is sized $z \log\log n$, and it must also be shuffled, the Bloom filter construction cost of $O(z \log z \log\log n)$ dominates asymptotically. Summing over the i levels sized $z = 4^i$, and amortizing over the queries for each level between rebuilding, we find a total amortized offline cost of $O(\log^2 n \log\log n)$

A query requires a single online round trip: the client generates a query object, sends it to the server, and receives a response containing the answer. The offline shuffle process requires several round trips (as discussed in Section 5.1), but this cost is amortized over a period corresponding to many queries, so that the average number of round trips per query is still well under 2.

We also estimate the cost of querying and shuffling for a sample hardware configuration. The *online* cost is now limited by network, disk, and encryption throughput (instead of, e.g., network round trips in related work). The *offline* cost is limited, as in existing work, by network, disk, and encryption throughput.

disk seek cost	0
total disk throughput	400 MBytes/sec
crypto throughput	200 MBytes/sec
net throughput	125 MBytes/sec
net round trip	50 ms

Figure 5: Assumed hardware config.

Acceptable failure rate	2^{-128}
Item block size	10,000 bytes
Symmetric key size	256 bits
Bloom filter hashes	50

Figure 6: Database parameters

To make the comparison between SR-ORAM and existing work as general as possible, we consider an *ideal storage-free interactive ORAM*, which includes just two core costs inextricably linked to any pyramidal storage-free ORAM, and ignores any other costs.

First is the online cost of retrieving an item from an ORAM interactively, requiring $\log_2 n$ round trips and the data transfer cost of $\log_2 n$ blocks. Second is the offline cost of obliviously constructing each level, assuming 2^i real items and 2^i fakes at each level i, using a randomized shell sort. Thus, this ideal storage-free interactive ORAM provides a lower bound for any existing ORAMs that do not assume super-logarithmic client storage. Existing work, including [6] and [7], fall into the category of a storage-free interactive ORAM, but have significant other costs, pushing their cost in fact much above this lower bound.

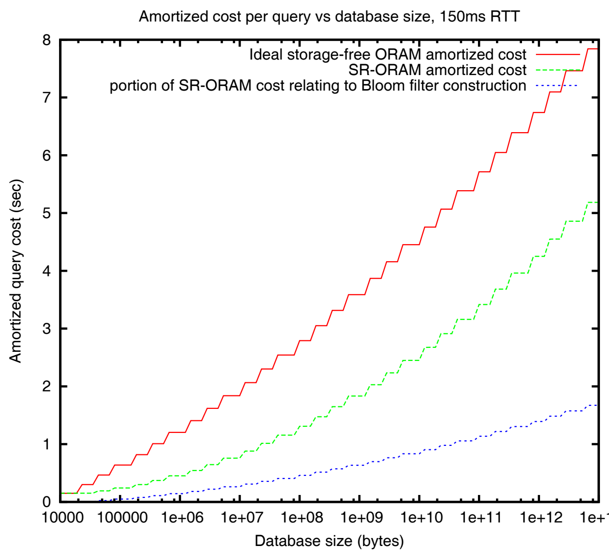
As discussed in Section 3.2, given an upper bound on acceptable failure (false positive) rate r, we can calculate the necessary Bloom filter size as a function of the number of hashes used for lookup k.

For example, for a Bloom filter false positive rate of $r = 2^{-64}$, and constant $k = 50$ (chosen to optimize a trade-off between the Bloom filter construction and online query costs—see Lemma 3), the resulting optimal Bloom filter size is under a third of the item storage size (of $2n$ blocks), for a wide variety of database sizes ranging from a megabyte up to a petabyte. For bounding the false positive rate at 2^{-128}, as we assume for the performance analysis below, Bloom filter storage of less than the total item storage size still suffices.

The target hardware configuration is listed in Figure 5, and the taret database configuration in Figure 6. Solid state disks are assumed, resulting in a low disk seek cost. Disk access time is modeled as a function of the disk throughput; divergence between this model and reality is examined and resolved in Section 7.1.

The results are illustrated in Figure 7. The item shuffle cost is unchanged over the ideal ORAM construction, but the Bloom filter shuffle cost adds a significant fraction to the overall level construction cost. As can be seen, however, the $\log_2 n$ round trips imposed by existing work quickly add up to exceed the SR-ORAM shuffle cost, in even moderate latency networks. In general, the additional offline cost is small compared to the savings in the online cost.

The SR-ORAM online cost encompasses the cost of transferring the query object across the network, reading each



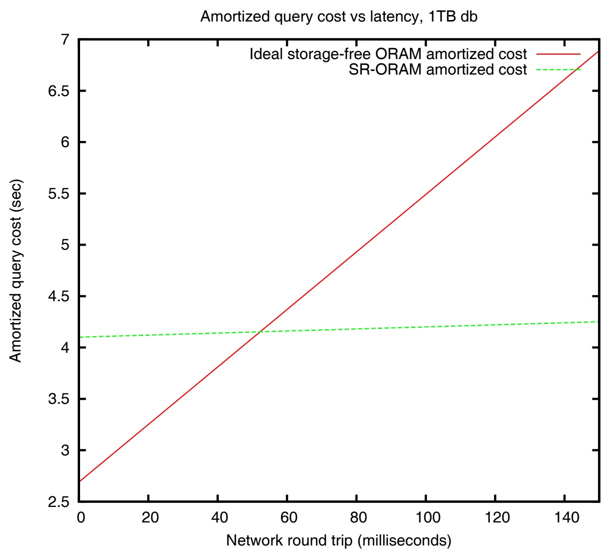


Figure 7: Comparison of amortized (online plus offline) cost of SR-ORAM to the ideal interactive storage-free ORAM, assuming the hardware configuration of Figure 5. Left: query cost plotted as the database size grows on a logarithmic X axis. Right: comparison vs. latency for a fixed database size.

key part from disk, server-side decryption, and transmitting the results back to the client. The offline performance cost encompasses the disk, encryption, and network costs of shuffling the items using a 4-pass random shell sort, and constructing the Bloom filter. The step shape results from the additional level shuffled and queried every time the database size doubles. While this no longer adds an additional round trip, it does increase the query object size, require reading an extra set of (e.g. 50) Bloom filter positions (key components) from disk, and require additional construction time.

Discussion. Offline shuffling in both the ideal storage-free ORAM and SR-ORAM takes somewhat longer than that described in [19] (who achieve over 1 query per second on a 1TB database) because we are eliminating their assumption of client storage. Instead of using a storage-based merge scramble that requires $\log \log n$ passes to sort n items, we use a randomized shell sort, that requires $\approx 24 \log n$ passes. This is not a result of our technique—storage can be applied to speed up our result equivalently—but a result of the different model here, under which we do not provide $O(\sqrt{n \log n})$ client storage.

Online vs. offline cost. In the preceeding analysis we consider the "online" cost to be the cost of performing an individual query, and the "offline" cost to be the cost of constructing the associated levels. However, the top level needs to be rebuilt in between each query. That could perhaps more appropriately be considered part of the online cost rather than the offline cost, resulting in a total of 1.5 round trips. This is because the client must write back the new version of the top level *after* each query. In the single-client usage model considered in this paper, however, this write-back can be included with the next item request, returning our online cost to only a single round trip per query.

The rest of the level constructions can also be performed in a small number of round trips, constant in the level size. De-amortization techniques are compatible with this ORAM, and eliminate the waiting during level shuffle. We recommend an appropriate de-amortization technique be applied to any implementation, but discussion is out of scope for this paper.

7.1 Dealing with Reality

Although it is convenient to model disk data transfer costs based only on the average disk throughput, this is not a complete model, not even for solid state disks. In the solid state disks we consider above, the disk sector size must be considered: it specifies the minimum size of a read or write we can perform. That is, reads or writes of smaller amounts cost the same as reading/writing the entire sector. This is problematic for one piece of the SR-ORAM cost analysis: the random shell sort of segments during the Bloom filter construction. The Bloom filter segments (e.g., 32 bytes) are potentially much smaller than a disk sector (e.g., 4096 bytes), and are accessed in a random pattern during the random shell sort.

We now describe implementation details that avoid the expense resulting from short disk reads. First, leading to a simple but expensive potential solution, recall that the disk costs are all server-side costs. This makes it plausible to solve this cost discrepancy with additional hardware. For example, a large amount of server RAM (e.g., 48 GB for a 1TB database) would make it possible to cache the *entire* Bloom filter during construction, requiring only a single contiguous write at the end. A key point is that the Bloom filter is smaller during these sort procedures than the final resulting Bloom filter will eventually be. For example, it is blown up by about a factor of 4 when converting the bits (with their, e.g., 64-bit positions) into (e.g., 256-bit) keys at the end. This conversion process requires only a single, sequential scan and write. This amount of RAM would result in a sort faster than the estimates used above (pushing the bottleneck to client crypto speeds instead of disk I/O speeds), which assume that disk transfer is required for each of the $\approx 24 \log_2 n$ passes across the data in the random shell sort. This is not necessarily an unreasonable scenario, as the RAM is only used for the duration of the sort, making it feasible to "pool" resources across many clients.

A second option is to assume enough client memory to build the Bloom filter locally. For a 1 TB database consisting of 10KB blocks, and taking the acceptable failure rate to be 2^{-128}, and 50 hash functions, the total number of Bloom filter positions to set can be under 16 billion bits. This fits in 2GB of client memory. Moreover, as this construction is now done in private client memory, the oblivious Bloom filter sort can be avoided, speeding up the Bloom filter construction significantly. This process now requires only the network transfer, and sequential disk write of the key-storing Bloom filter (.5TB). However, the client memory requirement for that (cost-optimal) Bloom filter construction process is linear in the total database size, a trait we desire avoiding.

Fortunately, such a workaround is not necessary. We now illustrate how the randomized shell sort can be performed without incurring the overhead resulting from the minimum sector size. In exchange, we require extra server-side sequential scans of the data. Observe that a randomized shell sort consists of dividing the array to be sorted (sized s) into equal sized regions, and swapping items between the two regions (called the "compareRegions" operation). The region sizes start at $s/2$ and decrease all the way down to 1. In any compareRegions operation, one region is read sequentially, while the other region is read randomly. Likewise, the regions are written back in the order they are read. Two properties are key to eliminating disk seek and partial sector read costs. First, observe that for regions that fit entirely in the server available cache sized M, the disk seek / minimum sector cost can be avoided altogether with an aggressive read-ahead page cache. This is because any one region can be read and cached in its entirety, and small regions are accessed contiguously.

Second, when the regions are too big to fit in the server page cache, the access pattern is still predictable by the server. This means it can sort data according to the future access pattern. Moreover, this sort can be performed in only $\log_M n$ passes. This is 2 passes whenever the server has $\sqrt{n}$ blocks of RAM, which we believe to be a more than reasonable assumption. The idea is, in one pass, sort data in groups sized M. In the second pass, merge these n/M regions together, reading contiguous chunks from each region into the page cache (the default behavior of a read-ahead page cache). This way, during the shell sort, the items being read can be accessed contiguously.

For simplicity, the data is sorted back into its start location after being written. The result is a sort that performs efficiently, even when the sort item size is much smaller than the disk sector size. The penalty is that now data is scanned from the disk up to four times in each sort pass (but this is more than made up for by the savings of not having to read an entire disk sector for every access of a 32-byte element).

This cost is reflected in the graphs above.

A similar argument comes into play when modeling the encryption/decryption throughput; our model considers the sustained throughput over large blocks of data, while when sorting the Bloom filter positions, many of the decryptions and encryptions are on short (e.g. 64-bit) blocks. Fortunately, the encryption block size (e.g., 256-bit) is much closer to the encryption size, resulting in the actual crypto throughput staying within a factor of 1/4 of the modeled crypto throughput. This analysis is also reflected in our graphs, which measure the entire cost of decrypting a cipher block even when the amount of data to be decrypted in a single operation is smaller than the block.

8. CONCLUSION

We introduced a new single-round-trip ORAM. We analyzed its security guarantees and demonstrated its utility. While non-interactivity is of significant theoretic interest in itself, we also showed this to be the most efficient Oblivious RAM to date for a storage-free client over today's Internet-scale network latencies.

9. ACKNOWLEDGEMENTS

This work was supported in part by the NSF under awards 1161541, 0937833, 0845192, 0803197, 0708025. The authors would also like to thank the anonymous reviewers for their excellent feedback.

10. REFERENCES

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

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

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

[4] Dan Boneh, David Maziéres, and Raluca Ada Popa. Remote oblivious storage: Making Oblivious RAM practical. Technical report, MIT, 2011. MIT-CSAIL-TR-2011-018 March 30, 2011.

[5] Ivan Damgård, Sigurd Meldgaard, and Jesper Nielsen. Perfectly secure Oblivious RAM without random oracles. In *Theory of Cryptography*, volume 6597 of *Lecture Notes in Computer Science*, pages 144–163. 2011.

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

[7] Michael Goodrich and Michael Mitzenmacher. Mapreduce parallel cuckoo hashing and Oblivious RAM simulations. In *38th International Colloquium on Automata, Languages and Programming (ICALP)*, 2011.

[8] Michael Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *ACM Cloud Computing Security Workshop at CCS (CCSW)*, 2011.

[9] Michael T. Goodrich. Randomized shellsort: A simple oblivious sorting algorithm. In *Proceedings 21st ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2010.

[10] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Practical oblivious storage. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, CODASPY '12, pages 13–24, New York, NY, USA, 2012. ACM.

[11] IBM. IBM 4764 PCI-X Cryptographic Coprocessor (PCIXCC). Online at `http://www-03.ibm.com/`

`security/cryptocards/pcixcc/overview.shtml`, 2006.

[12] 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.

[13] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 143–156. SIAM, 2012.

[14] Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In *CRYPTO*, pages 502–519, 2010.

[15] Sean W. Smith and David Safford. Practical server privacy with secure coprocessors. *IBM Systems Journal*, 40(3):683–695, 2001.

[16] Emil Stefanov, Elaine Shi, and Dawn Song. Towards Practical Oblivious RAM. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2012.

[17] 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.

[18] Peter Williams and Radu Sion. SR-ORAM: Single round-trip Oblivious RAM. In *Industrial Track of ACNS*, 2012.

[19] Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *ACM Conference on Computer and Communications Security (CCS)*, pages 139–148, 2008.