Path ORAM: An Extremely Simple Oblivious RAM Protocol

As Published: 10.1145/3177872

Publisher: Association for Computing Machinery (ACM)

Persistent URL: https://hdl.handle.net/1721.1/135810

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Path ORAM: An Extremely Simple Oblivious RAM Protocol

Emil Stefanov[†], Marten van Dijk[‡], Elaine Shi[*], Christopher Fletcher[◦], Ling Ren[◦], Xiangyao Yu[◦], Srinivas Devadas[◦]

[†] UC Berkeley [‡] UConn [*] UMD [◦] MIT CSAIL

ABSTRACT

We present Path ORAM, an extremely simple Oblivious RAM protocol with a small amount of client storage. Partly due to its simplicity, Path ORAM is the most practical ORAM scheme for small client storage known to date. We formally prove that Path ORAM requires $O(\log^2 N/\log \chi)$ bandwidth overhead for block size $B = \chi \log N$. For block sizes bigger than $\omega(\log^2 N)$, Path ORAM is asymptotically better than the best known ORAM scheme with small client storage. Due to its practicality, Path ORAM has been adopted in the design of secure processors since its proposal.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Algorithms, Security

Keywords

Oblivious RAM; ORAM; Path ORAM; access pattern

1. INTRODUCTION

It is well-known that data encryption alone is often not enough to protect users' privacy in outsourced storage applications. The sequence of storage locations accessed by the client (i.e., access pattern) can leak a significant amount of sensitive information about the unencrypted data through statistical inference. For example, Islam et. al. demonstrated that by observing accesses to an encrypted email repository, an adversary can infer as much as 80% of the search queries [21].

Oblivious RAM (ORAM) algorithms, first proposed by Goldreich and Ostrovsky [13], allow a client to conceal its access pattern to the remote storage by continuously shuffling and re-encrypting data as they are accessed. An adversary can observe the physical storage locations accessed, but the ORAM algorithm ensures that the adversary has negligible probability of learning anything about the true (logical) access pattern. Since its proposal, the research community has strived to find an ORAM scheme that is not only theoretically interesting, also practical [4, 7, 12, 14–17, 22, 23, 25–27, 30, 33–38].

In this paper, we propose a novel ORAM algorithm called *Path ORAM*[1]. This is to date the most practical ORAM construction under small client storage. We prove theoretical bounds on its performance and also present matching experimental results.

Path ORAM makes the following contributions:

Simplicity and practical efficiency. In comparison to other ORAM algorithms, our construction is arguably much simpler. Although we have no formal way of measuring its simplicity, the core of the Path ORAM algorithm can be described in just 16 lines of pseudocode (see Figure 1) and our construction does not require performing sophisticated deamortized oblivious sorting and oblivious cuckoo hash table construction like many existing ORAM algorithms [4, 7, 12–17, 22, 25–27, 35–37]. Instead, each ORAM access can be expressed as simply fetching and storing a single path in a tree stored remotely on the server. Path ORAM's simplicity makes it more practical than any existing ORAM construction with small (i.e., constant or poly-logarithmic) local storage.

Asymptotic efficiency. We prove that for a reasonably large block size $B = \chi \cdot \log N$ bits where N is the total number of blocks, Path ORAM with recursion (where recursion is proposed in Shi *et al.* [30]; see Section 3.7) achieves an asymptotic bandwidth cost of $O(\log^2 N/\log \chi)$ blocks, and consumes $O(\log^2 N/\log \chi)\omega(1)$ block client-side storage[2]. In other words, to access a single block, the client needs to in reality access $O(\log^2 N/\log \chi)$ physical blocks to hide its access patterns from the storage server. The above result achieves a failure probability of $N^{-\omega(1)}$, negligible in N.

As pointed out later in Section 1.1, our result outperforms the best known ORAM for small client storage [22], both in terms of asymptotics and practicality, for reasonably large block sizes, i.e., block sizes typically encountered in practical applications.

Practical and theoretic impact of Path ORAM. Since we first proposed Path ORAM [32] in February 2012, it

[1]Our construction is called Path ORAM because data on the server is always accessed in the form of tree paths.

[2]Throughout this paper, when we write the notation $g(n) = O(f(n)) \cdot \omega(1)$, we mean that for any function $h(n) = \omega(1)$, it holds that $g(n) = O(f(n)h(n))$.

CCS'13, November 4–8, 2013, Berlin, Germany.

ACM 978-1-4503-2477-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2508859.2516660.

ORAM Scheme	Client Storage (# blocks)	Read & Write Bandwidth (# blocks)
Kushilevitz *et al.* [22]	$O(1)$	$O(\log^2 N / \log\log N)$
Gentry *et al.* [11] ($B = \chi \log N$)	$O(\log^2 N) \cdot \omega(1)$	$O(\log^3 N / (\log\log N \log \chi)) \cdot \omega(1)$
Recursive Path ORAM ($B = \chi \cdot \log N$) (store stash on client)	$O(\log^2 N / \log \chi) \cdot \omega(1)$	$O(\log^2 N / \log \chi)$
Recursive Path ORAM ($B = \chi \cdot \log N$) (store stash on server)	$O(\log N) \cdot \omega(1)$	$O(\log^2 N / \log \chi) \cdot \omega(1)$

Table 1: Comparison to other ORAM schemes: Asymptotic bandwidth cost. B is the block size (in terms of # bits), N is the total number of blocks. The Path ORAM Stash is defined in Section 3. The failure probability is set to $N^{-\omega(1)}$ in this table, i.e., negligible in N.

has made both a practical and a theoretic impact in the community.

On the practical side, Path ORAM is the most suitable known algorithm for hardware ORAM implementations due to its conceptual simplicity, small client storage, and practical efficiency. Ren *et al.* built a simulator for an ORAM-enabled secure processor based on the Path ORAM algorithm [29] and the Ascend processor architecture [9, 10] uses Path ORAM as a primitive. Maas *et al.* [24] implemented Path ORAM on a secure processor using FPGAs and the Convey platform.

On the theoretic side, subsequent to the proposal of Path ORAM, several theoretic works adopted the same idea of path eviction in their ORAM constructions — notably the works by Gentry *et al.* [11] and Chung *et al.* [6]. These two works also try to improve ORAM bounds based on the binary tree construction by Shi *et al.* [30]; however, as pointed out in Section 1.1 our bound is asymptotically better than those by Gentry *et al.* [11] and Chung *et al.* [6]. Gentry's Path ORAM variant construction has also been applied to secure multiparty computation [11].

Novel proof techniques. Although our construction is simple, the proof for upper bounding the client storage is quite intricate and interesting. Our proof relies on the creation of a second ORAM construction and a reduction from Path ORAM to the second ORAM construction. We provide *concrete* bounds showing that for M load/store operations on N data blocks, recursive Path ORAM with client storage $\leq R \log N / \log \chi$ blocks, server storage $28N$ blocks and bandwidth $14(\log N)^2 / \log \chi$ blocks per load/store operation, fails during one of the M load/store operations with probability $\leq 14 \cdot 0.625^{-R} M \log N / \log \chi$. Our empirical results in Section 5 indicate that the constants in practice are even lower than our theoretic bounds.

1.1 Related Work

Oblivious RAM was first investigated by Goldreich and Ostrovsky [12, 13, 25] in the context of protecting software from piracy, and efficient simulation of programs on oblivious RAMs. Since then, there has been much subsequent work [4,6,7,11–15,17,22,25–27,35,37] devoted to improving ORAM constructions. Path ORAM is based upon the binary-tree ORAM framework proposed by Shi *et al.* [30].

Optimality of Path ORAM. Under small (i.e., constant or poly-logarithmic) client storage, the best known ORAM was proposed by Kushilevitz *et al.*, and has $O(\log^2 N / \log\log N)$ blocks bandwidth cost [22].

Our Path ORAM algorithm with recursion as in Shi *et al.* [30] is competitive with Kushilevitz *et al.* [22] in terms of bandwidth cost, when the block size is at least $\Omega(\log^2 N)$ bits; and can asymptotically outperform Kushilevitz *et al.* [22] for larger block sizes. For example, for block size $B = \Omega(\log^2 N)$ bits, our bandwidth cost is $O(\log^2 N / \log\log N)$ blocks, matching the best known bound of Kushilevitz *et al.* [22], under small client storage.

Of particular interest is the case when block size is at least $\Omega(\lambda)$ bits, where λ is the security parameter (e.g., $\lambda = 128$ or $\lambda = 256$) and $N = poly(\lambda)$ — since this is what one typically encounters in practical applications. In this case, recursive Path ORAM's bandwidth cost is only $O(\log N)$ blocks; moreover it has $O(1)$ round-trips since the depth of recursion would be constant. Goldreich and Ostrovsky show that under $O(1)$ client storage, any ORAM algorithm must have bandwidth overhead $\Omega(\log N)$ (regardless of the block size). Since then, a long-standing open question is whether it is possible to have an ORAM construction that has $O(1)$ or $poly\log(N)$ client-side storage and $O(\log N)$ blocks bandwidth cost [13, 14, 22]. Our bound partially addresses this open question for reasonably large block sizes.

Comparison with Gentry *et al.* and Chung *et al.*. Gentry *et al.* [11] improve on the binary tree ORAM scheme proposed by Shi *et al.* [30]. To achieve $2^{-\lambda}$ failure probability, their scheme achieves $O(\lambda(\log N)^2/(\log\lambda \log\chi))$ blocks bandwidth cost, for block size $B = \chi \cdot \log N$ bits. Assuming that $N = poly(\lambda)$, their bandwidth cost is $O(\lambda \log N / \log \chi)$ blocks. In comparison, recursive Path ORAM achieves $O(\log^2 N / \log \chi)$ blocks bandwidth cost. Note that typically $\lambda \gg \log N$ since $N = poly(\lambda)$. Therefore, recursive Path ORAM is much more efficient than the scheme by Gentry *et al.*. Table 1 presents this comparison, assuming a failure probability of $N^{-\omega(1)}$, i.e., negligible in N. Since $N = poly(\lambda)$, the failure probability can also equivalently be written as $\lambda^{-\omega(1)}$. We choose to use $N^{-\omega(1)}$ to simplify the notation in the asymptotic bounds.

Chung and Pass [6] proved a similar (in fact slightly worse) bound as Gentry *et al.* [11]. As mentioned earlier, our bound is asymptotically better than Gentry *et al.* [11] or Chung and Pass [6].

Very recently, Chung *et al.* proposed another statistically secure binary-tree ORAM algorithm [5] based on Path ORAM. Their theoretical bandwidth bound is $\log\log n$ factor worse than ours. Their simulation results suggest an empirical bucket size of 4 [1] — which means that their practical bandwidth cost is a constant factor worse than Path ORAM,

since they require operating on 3 paths in expectation for each data access, while Path ORAM requires reading and writing only 1 path.

Statistical security. We note that Path ORAM is also statistically secure (not counting the encryption). Statistically secure ORAMs have been studied in several prior works [2,8]. All known binary-tree based ORAM schemes and variants are also statistically secure [6, 11, 30] (assuming each bucket is a trivial ORAM).

2. PROBLEM DEFINITION

We consider a client that wishes to store data at a remote untrusted server while preserving its privacy. While traditional encryption schemes can provide confidentiality, they do not hide the data access pattern which can reveal very sensitive information to the untrusted server. In other words, the blocks accessed on the server and the order in which they were accessed is revealed. We assume that the server is untrusted, and the client is trusted, including the client's processor, memory, and disk.

The goal of ORAM is to completely hide the data access pattern (which blocks were read/written) from the server. From the server's perspective, read/write operations are indistinguishable from random requests.

Notations. We assume that the client fetches/stores data on the server in atomic units, referred to as *blocks*, of size B bits each. For example, a typical value for B for cloud storage is $64 - 256$ KB while for secure processors smaller blocks (128 B to 4 KB) are preferable. Throughout the paper, let N be the working set, i.e., the number of distinct data blocks that are stored in ORAM.

Simplicity. We aim to provide an extremely simple ORAM construction in contrast with previous work. Our scheme consists of only 16 lines of pseudo-code as shown in Figure 1.

Security definitions. We adopt the standard security definition for ORAMs from [34]. Intuitively, the security definition requires that the server learns nothing about the access pattern. In other words, no information should be leaked about: 1) which data is being accessed; 2) how old it is (when it was last accessed); 3) whether the same data is being accessed (linkability); 4) access pattern (sequential, random, etc); or 5) whether the access is a read or a write.

Definition 1 (Security definition). *Let*

$$\vec{y} := ((\mathsf{op}_M, \mathsf{a}_M, \mathsf{data}_M), \ldots, (\mathsf{op}_1, \mathsf{a}_1, \mathsf{data}_1))$$

denote a data request sequence of length M, where each op_i denotes a $\mathsf{read}(\mathsf{a}_i)$ or a $\mathsf{write}(\mathsf{a}_i, \mathsf{data})$ operation. Specifically, a_i denotes the identifier of the block being read or written, and data_i denotes the data being written. In our notation, index 1 corresponds to the most recent load/store and index M corresponds to the oldest load/store operation.

Let $A(\vec{y})$ denote the (possibly randomized) sequence of accesses to the remote storage given the sequence of data requests $\vec{y}$. An ORAM construction is said to be secure if (1) for any two data request sequences $\vec{y}$ and $\vec{z}$ of the same length, their access patterns $A(\vec{y})$ and $A(\vec{z})$ are computationally indistinguishable by anyone but the client, and (2) the ORAM construction is correct in that it returns on input $\vec{y}$ data that is consistent with $\vec{y}$ with probability $\geq 1 - negl(|\vec{y}|)$, i.e., the ORAM may fail with probability $negl(|\vec{y}|)$.

Like all other related work, our ORAM constructions do not consider information leakage through the timing channel, such as when or how frequently the client makes data requests. Achieving integrity against a potentially malicious server is discussed in Section 3.8. We do not focus on integrity in our main presentation.

3. THE PATH ORAM PROTOCOL

We first describe the Path ORAM protocol with $N/\chi + O(\log N) \cdot \omega(1)$ blocks of client storage, and then later in Section 3.7 we explain how the client storage can be reduced to $O(\log^2 N / \log \chi) \cdot \omega(1)$ blocks via recursion.

3.1 Overview

We now give an informal overview of the Path ORAM protocol. The client stores a small amount of local data in a stash. The server-side storage is treated as a binary tree where each node is a bucket that can hold up to a fixed number of blocks.

Main invariant. We maintain the invariant that at any time, each block is mapped to a uniformly random leaf bucket in the tree, and unstashed blocks are always placed in some bucket along the path to the mapped leaf.

Whenever a block is read from the server, the entire path to the mapped leaf is read into the stash, the requested block is remapped to another leaf, and then the path that was just read is written back to the server. When the path is written back to the server, additional blocks in the stash may be evicted into the path as long as the invariant is preserved and there is remaining space in the buckets.

3.2 Server Storage

Data on the server is stored in a tree consisting of buckets as nodes. The tree does not have to necessarily be a binary tree, but we use a binary tree in our description for simplicity.

Binary tree. The server stores a binary tree data structure of height $L = \lceil \log_2(N) \rceil - 1$ and 2^L leafs. The tree can easily be laid out as a flat array when stored on disk. The levels of the tree are numbered 0 to L where level 0 denotes the root of the tree and level L denotes the leafs.

Bucket. Each node in the tree is called a bucket. Each bucket can contain up to Z real blocks. If a bucket has less than Z real blocks, it is padded with dummy blocks to always be of size Z. It suffices to choose the bucket size Z to be a small constant such as $Z = 4$ (see Section 5.1).

Path. Let $x \in \{0, 1, \ldots, 2^L - 1\}$ denote the x-th leaf node in the tree. Any leaf node x defines a unique path from leaf x to the root of the tree. We use $\mathcal{P}(x)$ to denote set of buckets along the path from leaf x to the root. Additionally, $\mathcal{P}(x, \ell)$ denotes the bucket in $\mathcal{P}(x)$ at level ℓ in the tree.

Server storage size. Since there are about N buckets in the tree, the total server storage used is about $Z \cdot N$ blocks.

3.3 Client Storage and Bandwidth

The storage on the client consists of 2 data structures, a stash and a position map:

Stash. During the course of the algorithm, the client locally stores a small number of blocks in a local data structure S called the stash. In Section 6, we prove that the stash has a worst-case size of $O(\log N) \cdot \omega(1)$ blocks with high

N	Total # blocks outsourced to server
$L = \lceil \log_2 N \rceil - 1$	Height of binary tree
B	Block size (in bits)
Z	Capacity of each bucket (in blocks)
$\mathcal{P}(x)$	path from leaf node x to the root
$\mathcal{P}(x, \ell)$	the bucket at level ℓ along the path $\mathcal{P}(x)$
S	client's local stash
position	client's local position map
$x := \mathsf{position}[\mathsf{a}]$	block a is currently associated with leaf node x, i.e., block a resides somewhere along $\mathcal{P}(x)$ or in the stash.

Table 2: Notations.

probability. In fact, in Section 5.2, we show that the stash is usually empty after each ORAM read/write operation completes.

Position map. The client stores a position map, such that $x := \mathsf{position}[\mathsf{a}]$ means that block a is currently mapped to the x-th leaf node — this means that block a resides in some bucket in path $\mathcal{P}(x)$, or in the stash. The position map changes over time as blocks are accessed and remapped.

Bandwidth. For each load or store operation, the client reads a path of $Z \log N$ blocks from the server and then writes them back, resulting in a total of $2Z \log N$ blocks bandwidth used per access. Since Z is a constant, the bandwidth usage is $O(\log(N))$ blocks.

Client storage size. Note that the position map is of size $NL = N \log N$ bits, or equivalently, N/χ blocks with block size $B = \chi \cdot \log N$. In Section 3.7, we use this property to recursively store the position map in $\log N / \log \chi$ separate Path ORAMs. This reduces the client storage to $O(\log^2 N / \log \chi) \cdot \omega(1)$ at the cost of increasing the bandwidth to $O(\log^2 N / \log \chi)$.

If the client stores its storage at the server, then at every load/store operation the client needs to retrieve this storage from the server. Since the client accesses the $\log N / \log \chi$ separate ORAMs one after another, the client only needs sufficient storage for reading in a single path and stash of each ORAM separately: This leads to a client storage of $O(\log N) \cdot \omega(1)$ and bandwidth $O(\log^2 N / \log \chi) \cdot \omega(1)$.

3.4 Path ORAM Initialization

The client stash S is initially empty. The server buckets are intialized to contain random encryptions of the dummy block (i.e., initially no block is stored on the server). The client's position map is filled with independent random numbers between 0 and $2^L - 1$. The position map initially contains null for the position of every block. The position null is a special value indicating that the corresponding block has never been accessed and the client should assume it has a default value of zero.

3.5 Path ORAM Reads and Writes

In our construction, reading and writing a block to ORAM is done via a single protocol called Access described in Figure 1. Specifically, to read block a, the client performs $\mathsf{data} \leftarrow \mathsf{Access}(\mathsf{read}, \mathsf{a}, \mathsf{None})$ and to write data^* to block a, the client performs $\mathsf{Access}(\mathsf{write}, \mathsf{a}, \mathsf{data}^*)$. The Access protocol can be summarized in 4 simple steps:

$\mathsf{Access}(\mathsf{op}, \mathsf{a}, \mathsf{data}^*)$:

1: $x \leftarrow \mathsf{position}[\mathsf{a}]$
2: $\mathsf{position}[\mathsf{a}] \leftarrow \mathsf{UniformRandom}(0 \ldots 2^L - 1)$
3: **for** $\ell \in \{0, 1, \ldots, L\}$ **do**
4: $\quad S \leftarrow S \cup \mathsf{ReadBucket}(\mathcal{P}(x, \ell))$
5: **end for**
6: $\mathsf{data} \leftarrow$ Read block a from S
7: **if** $\mathsf{op} = \mathsf{write}$ **then**
8: $\quad S \leftarrow (S - \{(\mathsf{a}, \mathsf{data})\}) \cup \{(\mathsf{a}, \mathsf{data}^*)\}$
9: **end if**
10: **for** $\ell \in \{L, L-1, \ldots, 0\}$ **do**
11: $\quad S' \leftarrow \{(\mathsf{a}', \mathsf{data}') \in S : \mathcal{P}(x, \ell) = \mathcal{P}(\mathsf{position}[\mathsf{a}'], \ell)\}$
12: $\quad S' \leftarrow$ Select $\min(|S'|, Z)$ blocks from S'.
13: $\quad S \leftarrow S - S'$
14: $\quad \mathsf{WriteBucket}(\mathcal{P}(x, \ell), S')$
15: **end for**
16: **return** data

Figure 1: Protocol for data access. Read or write a data block identified by a. If $\mathsf{op} = \mathsf{read}$, the input parameter $\mathsf{data}^* = \mathsf{None}$, and the Access operation reads block a from the ORAM. If $\mathsf{op} = \mathsf{write}$, the Access operation writes the specified data^* to the block identified by a and returns the block's old data.

1. **Remap block** (Lines 1 to 2): Randomly remap the position of block a to a new random position. Let x denote the block's old position.
2. **Read path** (Lines 3 to 5): Read the path $\mathcal{P}(x)$ containing block a.
3. **Update block** (Lines 6 to 9): If the access is a write, update the data stored for block a.
4. **Write path** (Lines 10 to 15): Write the path back and possibly include some additional blocks from the stash if they can be placed into the path. Buckets are greedily filled with blocks in the stash in the order of leaf to root, ensuring that blocks get pushed as deep down into the tree as possible. A block a' can be placed in the bucket at level ℓ only if the path $\mathcal{P}(\mathsf{position}[\mathsf{a}'])$ to the leaf of block a' intersects the path accessed $\mathcal{P}(x)$ at level ℓ. In other words, if $\mathcal{P}(x, \ell) = \mathcal{P}(\mathsf{position}[\mathsf{a}'], \ell)$.

Subroutines. We now explain the ReadBucket and the WriteBucket subroutine. For $\mathsf{ReadBucket}(\mathsf{bucket})$, the client reads all Z blocks (including any dummy blocks) from the bucket stored on the server. Blocks are decrypted as they are read.

For $\mathsf{WriteBucket}(\mathsf{bucket}, \mathsf{blocks})$, the client writes the blocks blocks into the specified bucket on the server. When writing, the client pads blocks with dummy blocks to make it of size Z — note that this is important for security. All blocks (including dummy blocks) are re-encrypted, using a randomized encryption scheme, as they are written.

Computation. Client's computation is $O(\log N) \cdot \omega(1)$ per data access. In practice, the majority of this time is spent decrypting and encrypting $O(\log N)$ blocks per data access. We treat the server as a network storage device, so it only needs to do the computation necessary to retrieve and store $O(\log N)$ blocks per data access.

3.6 Security Analysis

To prove the security of Path-ORAM, let $\vec{y}$ be a data request sequence of size M. By the definition of Path-ORAM, the server sees $A(\vec{y})$ which is a sequence

$$\mathbf{p} = (\mathsf{position}_M[\mathsf{a}_M], \mathsf{position}_{M-1}[\mathsf{a}_{M-1}], \ldots, \mathsf{position}_1[\mathsf{a}_1]),$$

where $\mathsf{position}_j[\mathsf{a}_j]$ is the position of address a_j indicated by the position map for the j-th load/store operation, together with a sequence of encrypted paths $\mathcal{P}(\mathsf{position}_j(\mathsf{a}_j))$, $1 \leq j \leq M$, each encrypted using randomized encryption. The sequence of encrypted paths is computationally indistinguishable from a random sequence of bit strings by the definition of randomized encryption (note that ciphertexts that correspond to the same plaintext use different randomness and are therefore indistinguishable from one another). The order of accesses from M to 1 follows the notation from Definition 1.

Notice that once $\mathsf{position}_i(\mathsf{a}_i)$ is revealed to the server, it is remapped to a completely new random label, hence, $\mathsf{position}_i(\mathsf{a}_i)$ is statistically independent of $\mathsf{position}_j(\mathsf{a}_j)$ for $j < i$ with $\mathsf{a}_j = \mathsf{a}_i$. Since the positions of different addresses do not affect one another in Path ORAM, $\mathsf{position}_i(\mathsf{a}_i)$ is statistically independent of $\mathsf{position}_j(\mathsf{a}_j)$ for $j < i$ with $\mathsf{a}_j \neq \mathsf{a}_i$. This shows that $\mathsf{position}_i(\mathsf{a}_i)$ is statistically independent of $\mathsf{position}_j(\mathsf{a}_j)$ for $j < i$, therefore, (by using Bayes rule) $Prob(\mathbf{p}) = \prod_{j=1}^{M} Prob(\mathsf{position}_j(\mathsf{a}_j)) = (2^l)^{-M}$. This proves that $A(\vec{y})$ is computationally indistinguishable from a random sequence of bit strings.

Now the security follows from Theorem 1 in Section 6: For a stash size $O(\log N) \cdot \omega(1)$ Path ORAM fails (in that it exceeds the stash size) with at most negligible probability.

3.7 Recursion to Reduce Client Storage

For $\chi \geq 2$, we now explain how to reduce the client storage from N/χ blocks to $O((\log N)^2/\log \chi) \cdot \omega(1)$ blocks using an approach similar to the one introduced in [34]. The reduction of client storage comes at the cost of increasing the bandwidth from $O(\log N)$ to $O(\log^2 N/\log \chi)$. Notice that assuming that the block size $B = \chi \cdot \log(N)$ bits with $\chi \geq 2$ is a reasonable assumption that has been made by Stefanov *et al.* [33, 34] and Shi *et al.* [30]. For example, a standard 4KB block consists of 32768 bits and this assumption holds for all $N \leq 2^{16382}$.

The main idea is to recursively store the position map in a smaller Path ORAM with $N' = N/\chi$ blocks of size B bits. After $\log N/\log \chi$ recursions, this leads to a constant sized position map of the final Path ORAM. The position map stored on the client is only the one $O(1)$ sized position map for the last level of recursion. The client still needs to store the an $O(\log N) \cdot \omega(1)$ stash for each of the $O(\log N/\log \chi)$ levels of recursion, resulting in a total of $O((\log N)^2/\log \chi) \cdot \omega(1)$ blocks for the total client storage.

Path ORAM access with recursion. Consider a recursive Path ORAM made up of a series of ORAMs called $\mathsf{ORam}_0, \mathsf{ORam}_1, \mathsf{ORam}_2, \ldots, \mathsf{ORam}_X$ where ORam_0 contains the data blocks, the position map of ORam_i is stored in ORam_{i+1}, and the client stores the position map for ORam_X. To access a block in ORam_0, the client looks up its position in ORam_1, which triggers a recursive call to look up the position of the position in ORam_2, and so on until finally a position of ORam_X is looked up in the client storage. Essentially, we can replace lines 1–2 in Figure 1 with a recursive call to Access.

3.8 Integrity

Our protocol can be easily extended to provide integrity (with freshness) for every access to the untrusted server storage. Because data from untrusted storage is always fetched and stored in the form of a tree paths, we can achieve integrity by simply treating the Path ORAM tree as a Merkle tree where data is stored in all nodes of the tree (not just the leaf nodes). In other words, each node (bucket) of the Path ORAM tree is tagged with a hash of the following form

$$H(b_1 \parallel b_2 \parallel \ldots \parallel b_Z \parallel h_1 \parallel h_2)$$

where b_i for $i \in \{1, 2, \ldots, Z\}$ are the blocks in the bucket (some of which could be dummy blocks) and h_1 and h_2 are the hashes of the left and right child. For leaf nodes, $h_1 = h_2 = 0$. Hence only two hashes (for the node and its sibling) needs to be read or written for each ReadBucket or WriteBucket operation.

In [28], Ren *et al.* further optimize the integrity verification overhead for the recursive Path ORAM construction.

4. APPLICATIONS

4.1 Oblivious Binary Search Tree

Based on a class of recursive, binary tree based ORAM constructions, Gentry *et al.* propose a novel method for performing an entire binary search using a single ORAM lookup [11]. Their method is immediately applicable to Path ORAM. As a result, Path ORAM can be used to perform search on an oblivious binary search tree, using $O(\log^2 N/\log \chi)$ bandwidth. Note that since a binary search requires navigating a path of $O(\log N)$ nodes, using existing generic ORAM techniques would lead to bandwidth cost of $O((\log N)^3/\log\log N)$.

4.2 Stateless ORAM

Oblivious RAM is often considered in a single-client model, but it is sometimes useful to have multiple clients accessing the same ORAM. In that case, in order to avoid complicated (and possibly expensive) oblivious state synchronization between the clients, Goodrich *et al.* introduce the concept of *stateless* ORAM [18] where the client state is small enough so that any client accessing the ORAM can download it before each data access and upload it afterwards. Then, the only thing clients need to store is the private key for the ORAM (which does not change as the data in the ORAM changes).

In our recursive Path ORAM construction, we can download and upload the client state before and after each access. Since the client state is only $O(\log^2 N/\log \chi) \cdot \omega(1)$ and the bandwidth is $O(\log^2 N/\log \chi)$, we can reduce the permanent client state to $O(1)$ and achieve a bandwidth of $O(\log^2 N/\log \chi) \cdot \omega(1)$. Note that *during* an access the client still needs about $O(\log N) \cdot \omega(1)$ *transient* memory to perform the Access operation, but after the Access operation completes, the client only needs to store the private key.

4.3 Secure Processors

In a secure processor setting, private computation is done inside a tamper-resistant processor (or board) and main memory (e.g., DRAM) accesses are vulnerable to eavesdropping

and tampering. As mentioned earlier, Path ORAM is particularly amenable to hardware design because of its simplicity and low on-chip storage requirements.

Fletcher *et al.* [9, 10] and Ren *et al.* [29] built a simulator for a secure processor based on Path ORAM. They optimize the bandwidth cost and stash size of the recursive construction by using a smaller $Z = 3$ in combination with a background eviction process that does not break the Path ORAM invariant.

Maas *et al.* [24] built a hardware implementation of a Path ORAM based secure processor using FPGAs and the Convey platform.

Ren *et al.* [29] and Maas *et al.* [24] report about 1.2X to 5X performance overhead for many benchmarks such as SPEC traces and SQLite queries. To achieve this high performance, these hardware Path ORAM designs rely on on-chip caches while making Path ORAM requests only when last-level cache misses occur.

5. EVALUATION

5.1 Stash Occupancy Distribution

Stash occupancy. *In both the experimental results and the theoretical analysis, we define the stash occupancy to be the number of overflowing blocks (i.e., the number of blocks that remain in the stash) after the write-back phase of each ORAM access.* This represents the *persistent* local storage required on the client-side. In addition, the client also requires under $Z \log_2 N$ *transient* storage for temporarily caching a path fetched from the server during each ORAM access.

Our main theorem in Section 6 shows the probability of stash overflow decreases exponentially with the stash size, given that the bucket size Z is large enough. This theorem is verified by experimental results as shown in Figure 3 and Figure 2. In each experiment, the ORAM is initially empty. We first load N blocks into ORAM and then access each block in a round-robin pattern. I.e., the access sequence is $\{1, 2, \ldots, N, 1, 2, \ldots, N, 1, 2, \ldots\}$. Section 6.3 shows this is a worst-case access pattern in terms of stash occupancy for Path ORAM. We simulate our Path ORAM for a single run for about 250 billion accesses after doing 1 billion accesses for warming-up the ORAM. It is well-known that if a stochastic process is regenerative (empirically verified to be the case for Path ORAM), the time average over a single run is equivalent to the ensemble average over multiple runs (see Chapter 5 of [19]).

Figure 2 shows the minimum stash size to get a failure probability less than $2^{-\lambda}$ with λ being the security parameter on the x-axis. In Table 3, we extrapolate those results for realistic values of λ. The experiments show that the required stash size grows linearly with the security parameter, which is in accordance with the Main Theorem in Section 6 that the failure probability decreases exponentially with the stash size. Figure 3 shows the required stash size for a low failure probability ($2^{-\lambda}$) does not depend on N. This shows Path ORAM has good scalability.

Though we can only prove the theorem for $Z \geq 6$, in practice, $Z = 4$ and $Z = 5$ also work well. $Z = 3$ behaves relatively worse in terms of stash occupancy, and it is unclear whether $Z = 3$ works.

We only provide experimental results for small security parameters to show that the required stash size is $O(\lambda)$ and does not depend on N. Note that it is by definition infeasible to simulate for practically adopted security parameters (e.g., $\lambda = 128$), since if we can simulate a failure in any reasonable amount of time with such values, they would not be considered secure.

A similar empirical analysis of the stash size (but with the path included in the stash) was done by Maas *et al.* [24].

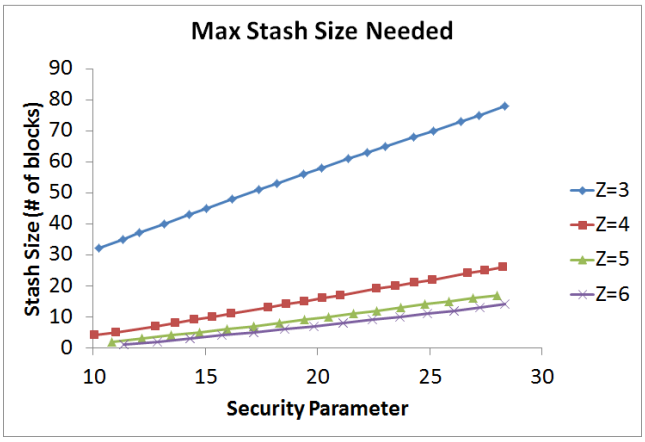


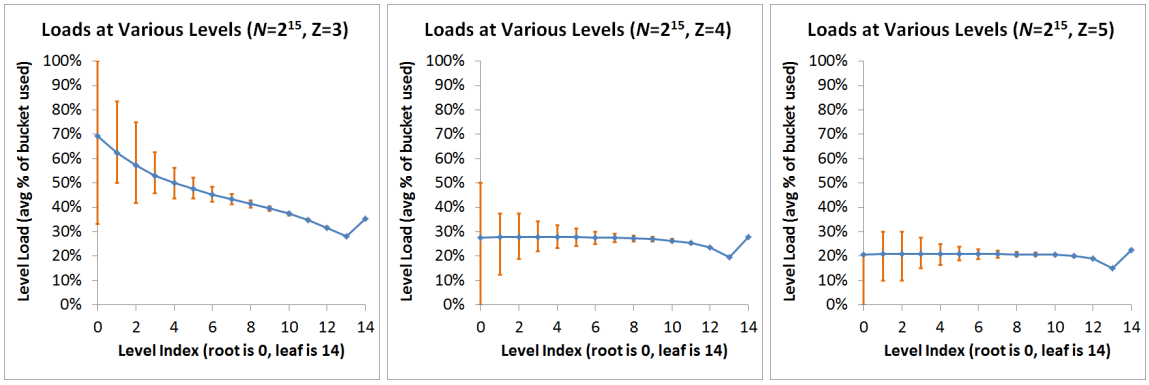
Figure 2: Empirical estimation of the required stash size to achieve failure probability less than $2^{-\lambda}$ where λ is the security parameter. Measured for $N = 2^{16}$, but as Figure 3 shows, the stash size does not depend on N (at least for $Z = 4$). The measurements represent a worst-case (in terms of stash size) access pattern. The stash size does not include the temporarily fetched path during Access.

Security Parameter (λ)	Bucket Size (Z)		
	4	5	6
	Max Stash Size		
80	89	63	53
128	147	105	89
256	303	218	186

Table 3: Required max stash size for large security parameters. Shows the maximum stash size required such that the probability of exceeding the stash size is less than $2^{-\lambda}$ for a worst-case (in terms of stash size) access pattern. Extrapolated based on empirical results for $\lambda \leq 26$. The stash size does not include the temporarily fetched path during Access.

5.2 Bucket Load

Figure 4 gives the bucket load per level for $Z \in \{3, 4, 5\}$. We prove in Section 6.3 that for $Z \geq 6$, the expected usage of a subtree T is close to the number of buckets in it. And Figure 4 shows this also holds for $4 \leq Z \leq 5$. For the levels close to the root, the expected bucket load is indeed 1 block (about 25% for $Z = 4$ and 20% for $Z = 5$). The fact that the root bucket is seldom full indicates the stash is empty after a path write-back most of the time. Leaves have slightly heavier loads as blocks accumulate at the leaves of the tree. $Z = 3$, however, exhibits a different distribution of bucket load (as mentioned in Section 5.1 and shown in Figure 2, $Z = 3$ produces much larger stash sizes in practice).



(a) Z=3 (b) Z=4 (c) Z=5

Figure 4: Average bucket load of each level for different bucket sizes. The error bars represent the 1/4 and 3/4 quartiles. Measured for a worst-case (in terms of stash size) access pattern.

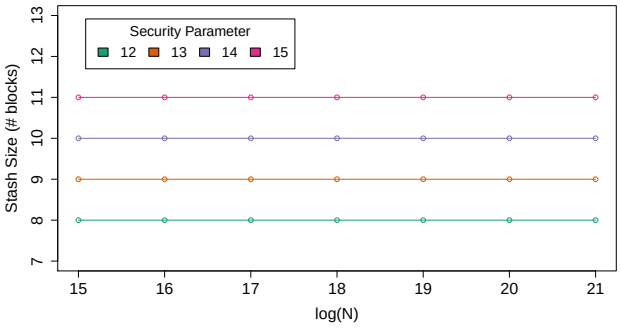


Figure 3: The stash size to achieve failure probability less than $2^{-\lambda}$ does not depend on N ($Z = 4$). Measured for a worst-case (in terms of stash size) access pattern. The stash size does not include the temporarily fetched path during Access.

6. THEORETIC BOUNDS

In this section we will bound the probability that, after a sequence of load/store operations, non-recursive Path-ORAM exceeds its stash size: We will show that this probability decreases exponentially in the size of the stash for a *constant* bucket size.

By $\mathcal{T}_L(Z)$ we denote a non-recursive Path-ORAM with $L+1$ levels in which each bucket stores Z real/dummy blocks; the root is at level 0 and the leafs are at level L.

We define a *sequence of load/store operations* $\mathbf{s}$ as a triple $(\mathbf{a}, \mathbf{x}, \mathbf{y})$ that contains (1) the sequence $\mathbf{a} = (a_j)_{j=1}^{s}$ of block addresses of blocks that are loaded/stored, (2) the sequence of labels $\mathbf{x} = (X_j)_{j=1}^{s}$ as seen by the server, and (3) the sequence $\mathbf{y} = (Y_j)_{j=1}^{s}$ of remapped leaf labels. (a_1, Y_1, X_1) corresponds to the most recent load/store operation, (a_2, Y_2, X_2) corresponds to the next most recent load/store operation, etc. The number of load/store operations is denoted by $s = |\mathbf{s}|$. The *working set* corresponding to $\mathbf{a}$ is defined as the number of distinct block addresses a_j in $\mathbf{a}$. We write $a(\mathbf{s}) = \mathbf{a}$.

By $\mathcal{T}_L(Z)[\mathbf{s}]$ we denote the distribution of real blocks in $\mathcal{T}_L(z)$ after a sequence $\mathbf{s}$ of load/store operations starting with an empty ORAM; $\mathbf{s}$ completely defines all the randomness needed to determine, for each block address a, its leaf label and which bucket/stash stores the block that corresponds to a. In particular the *number* of real blocks stored in the buckets/stash can be reconstructed.

We assume an infinite stash and in our analysis we investigate the usage $u(\mathcal{T}_L(Z)[\mathbf{s}])$ of the stash defined as the number of real blocks that are stored in the stash after a sequence $\mathbf{s}$ of load/store operations. In practice the stash is limited to some size R and Path-ORAM fails after a sequence $\mathbf{s}$ of load/store operations if the stash needs more space: this happens if and only if the usage of the infinite stash is at least $u(\mathcal{T}_L(Z)[\mathbf{s}]) > R$.

THEOREM 1 (MAIN). *Let $\mathbf{a}$ be any sequence of block addresses with a working set of size at most N. For a bucket size $Z = 7$, tree depth $L \geq \lceil \log N \rceil + 1$ and stash size R, the probability of a Path ORAM failure after a sequence of load/store operations corresponding to $\mathbf{a}$, is at most*

$$Prob(u(\mathcal{T}_L(Z)[\mathbf{s}]) > R | a(\mathbf{s}) = \mathbf{a}) \leq 14 \cdot 0.625^R,$$

where the probability is over the coin flips that determine $\mathbf{x}$ and $\mathbf{y}$ in $\mathbf{s} = (\mathbf{a}, \mathbf{x}, \mathbf{y})$.

For $Z = 6$ and $L \geq \lceil \log N \rceil + 3$ we obtain the bound $370 \cdot 0.667^R$.

As a corollary, for s load/store operations on N data blocks, Path ORAM with client storage $\leq R$ blocks, server storage $28N$ blocks and bandwidth $14 \log N$ blocks per load/store operation, fails during one of the s load/store operations with probability $\leq s \cdot 14 \cdot 0.625^R$. So, if we assume the number of load/stores is equal to $s = poly(N)$, then, for a stash of size $O(\log N)\omega(1)$, the probability of Path ORAM failure during one of the load/store operations is negligible in N.

Proof outline. The proof of the main theorem consists of several steps: First, we introduce a second ORAM, called ∞-ORAM, together with an algorithm that post-processes the stash and buckets of ∞-ORAM in such a way that if ∞-ORAM gets accessed by a sequence $\mathbf{s}$ of load/store operations, then post-processing leads to a distribution of real blocks over buckets that is exactly the same as the distribution as in Path ORAM after being accessed by $\mathbf{s}$.

Second, we characterize the distributions of real blocks over buckets in (a not post-processed) ∞-ORAM for which

post-processing leads to a stash usage $> R$. We show that the stash usage after post-processing is $> R$ if and only if there exists a subtree T for which its "usage" in ∞-ORAM is more than its "capacity". This means that we can use the union bound to upper bound $Prob(\mathcal{T}_L(Z)[u(\mathbf{s})] > R|a(\mathbf{s}) = \mathbf{a}\ )$ as a sum of probabilities over subtrees.

Third, we analyze the usage of subtrees T. We show how a mixture of a binomial and a geometric probability distribution expresses the probability of the number of real blocks that do not get evicted from T after a sequence $\mathbf{s}$ of load/store operations. By using the Chernoff bounding technique we prove the main theorem.

6.1 ∞-ORAM

We define ∞-ORAM, denoted by $\mathcal{R}_L(Z)$, as an ORAM that exhibits the same tree structure as Path-ORAM with $L+1$ levels but where each bucket has an *infinite* size. The interface of $\mathcal{R}_L(Z)$ operates as the interface of $\mathcal{T}_L(\infty)$ with the following distinguishing exception: Each bucket in $\mathcal{R}_L(Z)$ is partially stored in server storage and partially stored in the ORAM stash at the client; Z indicates the maximum number of blocks in a bucket that can be stored in server storage. If a bucket has more than Z real blocks, then the additional blocks are stored in the ∞-ORAM stash.

We define *post-processing*[3] $G(\mathcal{R}_L(Z)[\mathbf{s}])$ as a Greedy algorithm G that takes as input the state of $\mathcal{R}_L(Z)$ after a sequence $\mathbf{s}$ of load/store operations and repeats the following strategy:

1. Select a block in the stash that was not selected before. Suppose it has leaf label L and is stored in the bucket at level h on the path from the root to leaf L.
2. Find the highest level $i \leq h$ such that the bucket at level i on the path to leaf L stores $< Z$ blocks. If such a bucket exists, then use it to store the block. If it does not exist, then leave the block in the stash.

LEMMA 1. *The stash usage in a post-processed ∞-ORAM is exactly the same as the stash usage in Path-ORAM:*

$$u(G(\mathcal{R}_L(Z)[\mathbf{s}])) = u(\mathcal{T}_L(Z)[\mathbf{s}]).$$

Proof: We first notice that the order in which the Greedy strategy G processes blocks from the stash does not affect the number of real blocks in each bucket that are stored in server storage after post-processing: Suppose the Greedy strategy first processes a stash block b_1 with label X_1 stored in a bucket at level h_1 and suppose it finds empty space in server storage at level i_1; the Greedy strategy up to block b_1 has used up all the empty space in server storage allocated to the buckets along the path to the leaf with label X_1 at levels $i_1 < i \leq h_1$. Suppose next a stash block b_2 with label X_2 stored in a bucket at level h_2 finds empty space in server storage at level i_2; the Greedy strategy up to block b_2, which includes the post-processing of b_1, has used up all the empty space in server storage allocated to the buckets along the path to the leaf with label X_2 at levels $i_2 < i \leq h_2$. If we swap the post-processing of b_1 and b_2, then b_2 is able to find empty space in the bucket at level i_2, but may find empty space at a higher level since b_1 has not yet been processed. In this case, $i_2 < i_1$ and paths X_1 and X_2 intersect at least up to and including level i_1. This means that b_2 is stored at level i_1 and b_1 will use the empty space in server storage at level i_2. This means that the number of blocks that are stored in the different buckets after post-processing b_1 and b_2 is the same if b_1 is processed before or after b_2.

Now, we are able to show, by using induction in $|\mathbf{s}|$, that for each bucket b in a post-processed ∞-ORAM after a sequence $\mathbf{s}$ of load/store operations, the number of real blocks stored in b that are in server storage is equal to the number of blocks stored in the equivalent bucket in Path-ORAM after applying the same sequence $\mathbf{s}$ of load/store operations: The statement clearly holds for $|\mathbf{s}| = 0$ when both ORAMs are empty. Suppose it holds for $|\mathbf{s}| \geq 0$. Consider the next load/store operation to some leaf L'. After reading the path to leaf L' into the cache, Path-ORAM moves blocks from its cache/stash into the path from root to leaf L' according to algorithm $G(.)$. Therefore, postprocessing after the first $|\mathbf{s}|$ operations followed by the Greedy approach of Path ORAM that processes the $(|\mathbf{s}|+1)$-th operation is equivalent to post-processing after $|\mathbf{s}|+1$ operations where some blocks may be post-processed twice. Since the order in which blocks are post-processed does not matter, we may group together the multiple times a block b is being post-processed and this is equivalent to post-processing b exactly once as in $G(.)$. The number of real blocks in the stash is the same and this proves the lemma.

6.2 Usage/Capacity Bounds

To investigate which distributions of real blocks over buckets in a not post-processed ∞-ORAM $\mathcal{R}_L(Z)$ lead to a stash usage of $> R$ after post-processing, we start by analyzing distributions of blocks over subtrees: When we talk about a subtree T, we always implicitly assume that it is rooted at the root of the ORAM tree, i.e., T contains the root of the ORAM tree and all its nodes are connected to this root. We define $n(T)$ to be the total number of nodes in T. For ∞-ORAM we define the usage $u(\mathcal{R}_L(Z)[\mathbf{s}]; T)$ of T after a sequence $\mathbf{s}$ of load/store operations as the actual number of real blocks that are stored in the buckets of T (in ∞-ORAM blocks in a bucket are either in server storage or in the stash).

The following lemma characterizes the stash usage:

LEMMA 2. *The stash usage $u(G(\mathcal{R}_L(Z)[\mathbf{s}]))$ in post-processed ∞-ORAM is $> R$ if and only if there exists a subtree T in $\mathcal{R}_L(Z)$ such that $u(\mathcal{R}_L(Z)[\mathbf{s}]; T) > n(T)Z + R$.*

Proof: For a stash of size R, since post-processing can at most store $n(T)Z$ real blocks in T in server storage, we may interpret $n(T)Z+R$ as the *capacity* of T. Clearly, if the usage of T is more than the capacity of T, then post-processing leads to a stash usage of $> R$.

Suppose that $u(G(\mathcal{R}_L(Z)[\mathbf{s}])) > R$. Define T as the union of all paths from the root to a bucket b for which the buckets along the path in $G(\mathcal{R}_L(Z)[\mathbf{s}])$ each have Z real blocks stored in server storage. Then, $u(G(\mathcal{R}_L(Z)[\mathbf{s}]); T) > n(T)Z + R$. Also, if a real block in T or in the stash originated from a bucket b, then the Greedy approach of post-processing implies that the buckets on the path from where the block is stored in T to bucket b each store Z real blocks, therefore, $b \in T$. This shows that $u(\mathcal{R}_L(Z)[\mathbf{s}]; T) \geq u(G(\mathcal{R}_L(Z)[\mathbf{s}]); T)$, which finishes the proof of the lemma.

[3]Notice that the post-processing allows us to reason about Path-ORAM's stash size. It does not hide memory access patterns: ∞-ORAM + post-processing is only a tool for analysis, it is not an ORAM algorithm in itself.

Let $\mathbf{a}$ be a sequence of block addresses. As a corollary to Lemmas 1 and 2, we obtain

$$\begin{aligned} & Prob(u(\mathcal{T}_L(Z)[\mathbf{s}]) > R | a(\mathbf{s}) = \mathbf{a}) \\ = \; & Prob(u(G(\mathcal{R}_L(Z)[\mathbf{s}])) > R | a(\mathbf{s}) = \mathbf{a}) \\ = \; & Prob(\exists_T \; u(\mathcal{R}_L(Z)[\mathbf{s}]; T) > n(T)Z + R | a(\mathbf{s}) = \mathbf{a}) \\ \leq \; & \sum_T Prob(u(\mathcal{R}_L(Z)[\mathbf{s}]; T) > n(T)Z + R | a(\mathbf{s}) = \mathbf{a}), \end{aligned}$$

where the inequality follows from the union bound. The probabilities are over the coin flips that determines $\mathbf{s}$ given $a(\mathbf{s}) = \mathbf{a}$.

Since the number of ordered binary trees of size n is equal to the Catalan number C_n, which is $\leq 4^n$,

$$Prob(u(\mathcal{T}_L(Z)[\mathbf{s}]) > R | a(\mathbf{s}) = \mathbf{a}) \leq \quad (1)$$
$$\sum_{n \geq 1} 4^n \max_{T: n(T) = n} Prob(u(\mathcal{R}_L(Z)[\mathbf{s}]; T) > nZ + R | a(\mathbf{s}) = \mathbf{a}).$$

6.3 Chernoff Bound

We will upper bound the probability in the sum of the right-hand side of (1) for T with $n(T) = n$. To this purpose let $\mathbf{a} = (a_j)_{j=1}^s$ be a sequence of block addresses with working set size N, i.e., $\mathbf{a}$ has N different block addresses. Let $\mathbf{s} = (\mathbf{a}, \mathbf{y} = (Y_j)_{j=1}^s, \mathbf{x} = (X_j)_{j=1}^s)$ be a sequence of load/store operations. In this section we only provide a proof sketch, precise arguments are in Section 6.4.

For each block Y_j we wish to determine the probability that Y_j gets evicted from T. We first observe that Y_j never gets evicted if $Y_j \in V$, the set of leafs in T that are also leafs in the ORAM tree. Therefore, the expectation of the usage of T is at most

$$\begin{aligned} & E[u(\mathcal{R}_L(Z)[\mathbf{s}]; T) | a(\mathbf{s}) = \mathbf{a}] \leq \\ & E[\#Y_j \in V] + E[u(\mathcal{R}_L(Z)[\mathbf{s}]; T) | a(\mathbf{s}) = \mathbf{a}, \forall_j Y_j \notin V]. \end{aligned}$$

Since the probability that $Y_j \in V$ is equal to $|V|/2^L$,

$$E[\#Y_j \in V] = N|V|/2^L. \quad (2)$$

Next we observe that Y_j can only get evicted over paths to leafs with labels $X_j, X_{j-1}, \ldots, X_1$ (which correspond to the more recent load/store operations). Let u_j, as a function of the label values for $X_j, X_{j-1}, \ldots, X_1$, be the probability that Y_j is evicted from T given $Y_j \notin V$. In ∞-ORAM the worst-case address pattern $\mathbf{a}$ for a fixed working set size N does not duplicate block addresses, i.e., $s = N$ and as a consequence the labels in $(X_j)_{j=1}^N$ and $(Y_j)_{j=1}^N$ are all statistically independent of one another: Assuming non-duplicated block addresses, $u_1 \leq u_2 \leq u_3 \leq \ldots$, hence, there exist segments $U_1 = \{1, 2, \ldots, q_1\} \neq \emptyset$, $U_2 = \{q_1 + 1, \ldots, q_1 + q_2\} \neq \emptyset$, ..., such that within each segment U_k the probabilities u_j, $j \in U_k$, are all equal to one another. I.e, there exists a probability h_k such that $h_k = u_j$, $j \in U_k$. We notice that the number of segments U_k with $h_k < 1$ equals $n - 2|V|$, the number of buckets that are one edge away from T, minus 1. By the definition of h_k,

$$E[\#Y_j, j \in U_k, \text{ not evicted from } T \mid |U_k|] = (1 - h_k)|U_k|. \quad (3)$$

We notice that $u_{j+1} \neq u_j$ if and only if X_{j+1} matters in the calculation of u_{j+1}, i.e., there exists a block that can get evicted from T over the path to the leaf with label X_{j+1} while it cannot get evicted over a path to a leaf with label X_i, $1 \leq i \leq j$. This means that $X_{j+1} \notin V$ and, for each

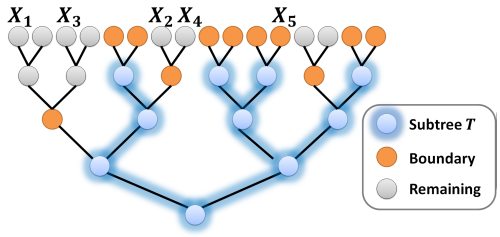


Figure 5: Subtree T for proof in Section 6.4.

$1 \leq i \leq j$, the intersection of the path to X_{j+1} and the path to X_i lies inside T, i.e., a block with label X_{j+1} cannot get evicted from T over a path to leaf X_i. This proves

$$\begin{aligned} & Prob(X_{j+1} \text{ such that } u_{j+1} = u_j) \\ = \; & Prob(X_{j+1} \in V \text{ or} \\ & \quad X_{j+1} \text{ can be evicted from } T \text{ over } (X_i)_{i=1}^j) \\ = \; & Prob(X_{j+1} \text{ evicted from } T | X_{j+1} \notin V) Prob(X_{j+1} \notin V) \\ & + Prob(X_{j+1} \in V) \\ = \; & u_j(1 - |V|/2^L) + |V|/2^L \\ = \; & 1 - (1 - u_j)(1 - |V|/2^L). \end{aligned}$$

This shows that $|U_k|$ has a geometric distribution with expectation $E[|U_k|]$ equal to

$$\begin{aligned} & \sum_{q \geq 1} (1 - (1 - h_k)(1 - |V|/2^L))^{q-1} (1 - h_k)(1 - |V|/2^L) q \\ & = \frac{1}{(1 - h_k)(1 - |V|/2^L)}. \end{aligned} \quad (4)$$

Combining (3) and (4) yields

$$\begin{aligned} & E[u(\mathcal{R}_L(Z)[\mathbf{s}]; T) | a(\mathbf{s}) = \mathbf{a}, \forall_j Y_j \notin V] \\ & \leq \sum_{k=1}^{n-2|V|} \frac{1 - h_k}{(1 - h_k)(1 - |V|/2^L)} = \frac{n - 2|V|}{1 - |V|/2^L}. \end{aligned}$$

Together with (2), the expectation of the usage of T is at most $N|V|/2^L + (n - 2|V|)/(1 - |V|/2^L)$.

In (1) we are only interested in a non-zero probability that T's usage is $> nZ + R$. Therefore, since $|V| \leq n$ and T's usage is at most N, we may constrain $|V| \leq N/Z$. For $N/2^L \leq 2$ and $Z \geq 4$, the expectation is at most $n/(1 - |V|/2^L) \leq 2n$. In the sum of the right-hand side of (1) we bound the probability that T's usage is $> nZ + R$, which is a multiple larger than the expectation of the usage in T. This means that we are able to apply a Chernoff bounding technique to prove the main theorem: For bucket size Z large enough, we are able to compensate the 4^n term in (1).

6.4 Details of Chernoff Bound

Before getting into proof details, we start by giving some intuition on how blocks get evicted from a subtree T in ∞-ORAM. For simplicity we assume T has no leafs with the ORAM tree in common. Figure 5 gives an example of a subtree T; the figure also shows the boundary nodes that are at one edge distance from T. When blocks get evicted from T, they will pass through one of these boundary nodes to a higher level outside T. X_1 is the most recent label of a block that was read from ORAM and written back with a new random label Y_1. X_2 is the next older label of a block

that was read from ORAM and written back with a new random label Y_2, etc.

We define S_j as the set of leafs of the ORAM tree that can be reached through a path that has a boundary node in common with one of the paths to X_1, X_2, ..., or X_j. In the example: $|S_1| < |S_2|$, since X_1 and X_2 have no boundary point in common; $S_2 = S_3$, since X_3 and X_1 have a boundary point in common; $S_3 = S_4$, since X_4 and X_2 have a boundary point in common; $|S_4| < |S_5|$, since X_5 has no boundary point in common with S_4.

Notice that the block with label Y_j only gets evicted if it is in S_j, i.e., one of the more recent load/stores of a path toX_i, $1 \leq i \leq j$, has a boundary node in common with the path to Y_j over which the corresponding block will get evicted from T. We introduce the following notation: $w_1 = |S_1| < w_2 = |S_2| = |S_3| = |S_4| < w_3 = |S_5| \ldots$ and $q_1 = 1$ denoting the number of sets S_j with size equal to w_1, $q_2 = 3$ denoting the number of sets S_j with size equal to w_2 etc. Let $m_1 = I(Y_1 \notin S_1)$, $m_2 = I(Y_2 \notin S_2) + I(Y_3 \notin S_3) + I(Y_4 \notin S_4)$, where $I(.)$ is the indicator function outputting 1 if the input is true and 0 if the input is false. We are interested in the sum $M = \sum_j m_j$, which is the number of blocks not evicted from T, in other words it equals the usage of T.

The probability distribution of q_i is a geometric distribution with its parameter a function of w_i. The probability distribution of m_i is a binomial distribution with its parameters characterized by q_i and w_i. The probability distribution of w_i is not correlated with q_i and m_i. These dependencies allow us to use Bayes rule to compute the probability distribution on M. The proof now proceeds by only considering the worst case sequence $(w_i)_{i \geq 1}$ and use probability generating functions to upper bound the tail distribution of M by using Chernoff bounding techniques.

We will now prove in detail an upper bound on the probability in the sum of the right-hand side of (1) for general T with $n(T) = n$. To this purpose let $\mathbf{a} = (a_j)_{j=1}^s$ be a sequence of block addresses with working set size N, i.e., $\mathbf{a}$ has N different block addresses.

Worst-case address pattern. In ∞-ORAM a worst-case address pattern $\mathbf{a}$ for a fixed working set size N does not duplicate block addresses, i.e., $s = N$ and as a consequence the labels in $(X_j)_{j=1}^N$ and $(Y_j)_{j=1}^N$ (that define $\mathbf{s}$ given $a(\mathbf{s}) = \mathbf{a}$) are all statistically independent of one another:

Suppose that there exists an address in $\mathbf{a}$ that has been loaded/stored twice in ∞-ORAM. Then, there exist indices i and j, $i < j$, with $a_i = a_j$. Without the j-th load/store, the working set remains the same and it is more likely for older blocks corresponding to a_k, $k > j$ to *not* have been evicted from T (since there is one less load/store that could have evicted an older block to a higher level outside T; also notice that buckets in ∞-ORAM are infinitely sized, so, removing the j-th load/store does not generate extra space that can be used for storage of older blocks that otherwise would not have found space). So, to maximize the probability of the right-hand side of (1), we may assume that $\mathbf{a} = (a_j)_{j=1}^s$ is a sequence of block addresses without duplicates.

Blocks that remain in T with probability 1. Let V be the set of leafs in T that are also leafs in the ORAM tree. Notice that if a block corresponds to a $Y_j \in V$, then it never gets evicted from T since the whole path to the leaf with label Y_j is part of T. The probability that m_0 out of the N labels Y_j are in V is equal to the binomial

$$p_0 = Prob(m_0) = \binom{N}{m_0} (\frac{|V|}{2^L})^{m_0} (1 - \frac{|V|}{2^L})^{N - m_0}. \quad (5)$$

T's usage in (1) is *upper bounded* by the number m_0 of Y_j's that are in V added to T's usage after an *infinite* sequence of load/store operations, where (1) all $Y_j \notin V$ (those Y_j that were in V are replaced by labels not in V and the sequence is extended with labels not in V) and (2) labels X_j and Y_j are uniformly distributed and statistically independent.

In (1) we are only interested in a non-zero probability that T's usage is $> nZ + R$. Therefore, without loss of generality, we may constrain $|V|$ to lead to such a non-zero probability. Since $|V| \leq n$ and T's usage is at most the number N of different blocks, we may assume $|V|Z \leq nZ + R \leq N$, hence,

$$|V| \leq N/Z. \quad (6)$$

Block eviction from T. Consider the infinite sequence of load/store operations. For each leaf label X we define $X(T)$ as the set of leafs that can be reached from a bucket $b \notin T$ that is on the path from the root to X. Notice that $X(T) = \emptyset$ if and only if $X \in V$. Furthermore, the number of different non-empty sets $X(T)$ is equal to the number of nodes in the ORAM tree that are one edge away from T:

$$\#\text{ distinct non-empty sets } X(T) = 1 + n - 2|V|. \quad (7)$$

The non-empty sets $X(T)$ do not intersect with V.

By the definition of $X_i(T)$, if ∞-ORAM reads/writes a path to X_i and $Y_j \in X_i(T)$ for $j \geq i$, then Y_j gets written to a bucket outside T: The block corresponding to Y_j is counted in T's usage if and only if $Y_j \notin S_j = \cup_{i=1}^j X_i(T)$.

Eviction probability as a function of the sizes $|S_j|$. The sizes of sets S_j are uniquely determined by the sequence

$$w_1 = |S_1| = \ldots = |S_{q_1}|,\ w_2 = |S_{q_1+1}| = \ldots = |S_{q_1+q_2}|, \text{ etc.}$$

By (7), the number of w_j is at most $1 + n - 2|V|$ and if $w_{1+n-2|V|}$ exists, then its is equal to $2^L - |V|$.

Let m_k be equal to the number of blocks in the subsequence $(Y_{(\sum_{t=1}^{k-1} q_t)+1}, \ldots, Y_{\sum_{t=1}^{k} q_t})$ that are $\notin S_{\sum_{t=1}^{k-1} q_t + 1} = \ldots = S_{\sum_{t=1}^{k} q_t}$, in other words, they do not get evicted from T. Since leaf labels Y_j are $\notin V$, the probability that $Y_j \notin S_j$ is equal to $1 - |S_j|/(2^L - |V|)$, hence,

$$Prob(m_k|(w_t, q_t, m_t)_{t=1}^{k-1}, w_k, q_k) = Prob(m_k|w_k, q_k)$$
$$= \binom{q_k}{m_k} (1 - \frac{w_k}{2^L - |V|})^{m_k} (\frac{w_k}{2^L - |V|})^{q_k - m_k}. \quad (8)$$

Probability (8) is zero for $k \geq 1 + n - 2|V|$, so we may restrict k to $\leq n - 2|V|$.

Since leaf labels X_j are not constrained, the probability that S_j has size $|S_j| = |S_{j-1}|$ is equal to $(|S_{j-1}| + |V|)/2^L$, the probability that $X_j \in S_{j-1} \cup V$. Hence,

$$Prob(q_k|(w_t, q_t, m_t)_{t=1}^{k-1}, w_k) = Prob(q_k|w_k)$$
$$= (\frac{w_k + |V|}{2^L})^{q_k - 1} (1 - \frac{w_k + |V|}{2^L}). \quad (9)$$

The probability that w_k takes on a certain value only depends on the previous values w_t, $t < k$, since they determine the sizes of previously added $X_j(T)$s and therefore determine the sizes of sets $X_j(T)$ that can still be added to S_{k-1}:

$$Prob(w_k|(w_t, q_t, m_t)_{t=1}^{k-1}) = Prob(w_k|(w_t)_{t=1}^{k-1}). \quad (10)$$

We combine (8), (9) and (10) by using Bayes rule twice:

$$\begin{aligned}&Prob((w_t,q_t,m_t)_{t\geq 1})\\&=\prod_{k\geq 1} Prob(m_k|w_k,q_k)Prob(q_k|w_k)Prob(w_k|(w_t)_{t=1}^{k-1})\\&=Prob((w_t)_{t\geq 1})\prod_{k\geq 1} Prob(m_k|w_k,q_k)Prob(q_k|w_k).\end{aligned}$$

If we sum over all possible sequences $(w_t,q_t)_{t\geq 1}$ we obtain

$$\begin{aligned}&Prob((m_t)_{t\geq 1})\\&=\sum_{(w_t,q_t)_{t\geq 1}} Prob((w_t)_{t\geq 1})\cdot\\&\quad\cdot\prod_{k\geq 1} Prob(m_k|w_k,q_k)Prob(q_k|w_k)\\&=\sum_{(w_t)_{t\geq 1}} Prob((w_t)_{t\geq 1})\cdot\\&\quad\cdot\prod_{k\geq 1}\sum_{q\geq 1} Prob(m_k|w_k,q_k=q)Prob(q_k=q|w_k).\end{aligned}$$

Substituting (8) and (9) yields

$$\begin{aligned}&Prob((m_t)_{t\geq 1})=\sum_{(w_t)_{t\geq 1}} Prob((w_t)_{t\geq 1})\prod_{k\geq 1} p_k \text{ where}\\&p_k=\sum_{q\geq 1}(\frac{w_k+|V|}{2^L})^{q-1}(1-\frac{w_k+|V|}{2^L})\cdot\\&\binom{q}{m_k}(1-\frac{w_k}{2^L-|V|})^{m_k}(\frac{w_k}{2^L-|V|})^{q-m_k}.\end{aligned}\tag{11}$$

Expected number of blocks in T. Notice that the expectation $E[m_k|(w_t)_{t\geq 1}]$ is equal to

$$\begin{aligned}&\sum_{q\geq 1}(\frac{w_k+|V|}{2^L})^{q-1}(1-\frac{w_k+|V|}{2^L})q(1-\frac{w_k}{2^L-|V|})\\&=\frac{1-w_k/(2^L-|V|)}{1-(w_k+|V|)/2^L}=1/(1-|V|/2^l).\end{aligned}$$

This is proved in a different way in Section 6.3, where we use (6) to show that the expectation of the usage of T is at most $2n$ for utilization $N/2^L\leq 2$ and for bucket size $Z\geq 4$. For this reason we expect to be able to use a Chernoff bound to prove that with negligible probability T's usage is more than T's capacity:

Probability generating functions. From (5) and (11):

$$\begin{aligned}&Prob(\sum_{t=0}^{n-2|V|} m_t>nZ+R)\leq \max_{(w_t)_{t\geq 1}} p, \text{ where}\\&p=\sum_{(m_t)_{t=0}^{n-2|V|} \text{ with } \sum_{t=0}^{n-2|V|} m_t>nZ+R} p_0\cdot\prod_{k\geq 1} p_k.\end{aligned}$$

The probability generating function corresponding to p_0 is

$$\begin{aligned}&\sum_{m=0}^{N}\binom{N}{m}(\frac{|V|}{2^L})^m(1-\frac{|V|}{2^L})^{N-m}X^m\\&=(1+(|V|/2^L)(X-1))^N\leq exp((|V|/2^L)(X-1)N)\end{aligned}$$

and the probability generating function corresponding to p_k is equal to

$$\sum_{q\geq 1}\alpha^{q-1}(1-\alpha)[1+\beta(X-1)]^q.\tag{12}$$

where

$$\alpha=\frac{w_k+|V|}{2^L}\leq 1 \text{ and } \beta=1-\frac{w_k}{2^L-|V|}.$$

By the binomial theorem, (12) is equal to

$$\begin{aligned}&\sum_{q\geq 1}\alpha^{q-1}(1-\alpha)\sum_{m=0}^{q}\binom{q}{m}(\beta(X-1))^m\\&=1+\sum_{m\geq 1}\sum_{q\geq m}\binom{q}{m}\alpha^{q-1}(1-\alpha)(\beta(X-1))^m\\&=1+\sum_{m\geq 1}\frac{\alpha^{m-1}}{(1-\alpha)^m}(\beta(X-1))^m.\end{aligned}$$

Notice that $\alpha\leq 1$ and $\beta/(1-\alpha)=1/(1-|V|/2^L)\leq 1/(1-N/(Z2^L))$ by (6).[4] This proves that (12) is at most

$$\begin{aligned}&1+\sum_{m\geq 1}\left[\frac{(X-1)}{1-N/(Z2^L)}\right]^m\\&=1+\frac{(X-1)}{(1-N/(Z2^L))-(X-1)}\\&\leq exp(\frac{(X-1)}{(1-N/(Z2^L))-(X-1)})\end{aligned}$$

for $1\leq X<2-N/(Z2^L)$.

The probability generating function (corresponding to p) of the total number of blocks $M=\sum_k m_k$ in T (T's usage) given $(w_t)_{t\geq 1}$ is equal to the product of the probability generating functions of p_j, $0\leq j\leq n-2|V|$, which is at most

$$exp(\frac{(X-1)}{(1-N/(Z2^L))-(X-1)}(n-2|V|)+(X-1)|V|\frac{N}{2^L}).$$

Let

$$L\geq\lceil\log N\rceil-1.\tag{13}$$

Then $N/2^L\leq 2$, hence, the contribution to $(X-1)|V|$ in the exponential is negative and we obtain the upper bound

$$exp(\frac{(X-1)}{(1-N/(Z2^L))-(X-1)}n).$$

Let $M[X]$ denote this function.

Markov's inequality. By Markov's inequality, $\forall z$ with

$$1<e^z<2-N/(Z2^L),\tag{14}$$

the tail distribution of the total number of blocks M in T given $(w_t)_{t\geq 1}$ is upper bounded by

$$\begin{aligned}Prob(M\geq nZ+R)&=Prob(e^{zM}\geq e^{z(nZ+R)})\\&\leq\frac{E[e^{zM}]}{e^{z(nZ+R)}}=\frac{M[e^z]}{e^{z(nZ+R)}}.\end{aligned}$$

[4] If we do not use the bound $\alpha\leq 1$ but explicitly maximize over all possible sequences $(w_t)_{t\geq 1}$ that may correspond to a subtree T of size n, then we may be able to prove the main theorem for even smaller bucket sizes $Z=4$ or $Z=5$.

Since the upper bound is independent of $(w_t)_{t\geq 1}$, it holds for the total number of blocks in T without a condition on $(w_t)_{t\geq 1}$, i.e.,

$$Prob(u(\mathcal{R}_L(Z)[\mathbf{s}];T) > nZ + R|a(\mathbf{s}) = \mathbf{a}) \leq \frac{M[e^z]}{e^{z(nZ+R)}}$$
$$= exp(n\left[\frac{e^z - 1}{2 - N/(Z2^L) - e^z} - zZ\right] - zR).$$

Main theorem. The first part of the main theorem follows by substituting $Z = 7$, $e^z = 1.6$ and bounding $L \geq \lceil \log N \rceil + 1$ (notice that both (13) and (14) hold), which gives $0.625^R/4.3^n$ as an upper bound on the tail distribution. Plugging this into the union bound (1) yields

$$Prob(u(\mathcal{T}_L(Z)[\mathbf{s}]) > R|a(\mathbf{s}) = \mathbf{a})$$
$$\leq \sum_{n\geq 1} 4^n 0.625^R/4.3^n \leq 14 \cdot 0.625^R. \qquad (15)$$

By substituting $Z = 6$, $e^z = 1.5$ and bounding $L \geq \lceil \log N \rceil + 3$ we obtain the bound $370 \cdot 0.667^R$, the second part of the main theorem.

Acknowledgments

This work is partially supported[5] by the NSF Graduate Research Fellowship grants DGE-0946797 and DGE-1122374, the DoD NFSEG Fellowship, NSF grant CNS-1314857, DARPA CRASH program N66001-10-2-4089, and a grant from the Amazon Web Services in Education program. We would like to thank Kai-Min Chung and Jonathan Katz for helpful discussions, Hubert Chan for his generous and unconditional help, and Kai-Min Chung for pointing out that our algorithm is statistically secure.

7. REFERENCES

[1] Personal communication with Kai-Min Chung, 2013.
[2] M. Ajtai. Oblivious rams without cryptogrpahic assumptions. In *Proceedings of the 42nd ACM symposium on Theory of computing*, STOC '10, pages 181–190, 2010.
[3] D. Asonov and J.-C. Freytag. Almost optimal private information retrieval. In *PET*, 2003.
[4] D. Boneh, D. Mazieres, and R. A. Popa. Remote oblivious storage: Making oblivious RAM practical. Manuscript, `http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf`, 2011.
[5] K.-M. Chung, Z. Liu, and R. Pass. Statistically-secure oram with $\tilde{O}(\log^2 n)$ overhead. `http://arxiv.org/abs/1307.3699`, 2013.
[6] K.-M. Chung and R. Pass. A simple oram. `https://eprint.iacr.org/2013/243.pdf`, 2013.
[7] I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious RAM without random oracles. In *TCC*, 2011.
[8] I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious ram without random oracles. In *TCC*, pages 144–163, 2011.
[9] C. Fletcher, M. van Dijk, and S. Devadas. Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *Proceedings of the 7th ACM CCS Workshop on Scalable Trusted Computing*, pages 3–8, Oct. 2012.
[10] C. W. Fletcher. Ascend: An architecture for performing secure computation on encrypted data. In *MIT CSAIL CSG Technical Memo 508*, April 2013.
[11] C. Gentry, K. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs. Optimizing oram and using it efficiently for secure computation. In *PETS*, 2013.
[12] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *STOC*, 1987.
[13] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 1996.
[14] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, 2011.
[15] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, pages 95–100, New York, NY, USA, 2011. ACM.
[16] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Practical oblivious storage. In *CODASPY*, 2012.
[17] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, 2012.
[18] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious ram simulation. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 157–167. SIAM, 2012.
[19] M. Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Performance Modeling and Design of Computer Systems: Queueing Theory in Action. Cambridge University Press, 2013.
[20] A. Iliev and S. W. Smith. Protecting client privacy with trusted computing at the server. *IEEE Security and Privacy*, 3(2):20–28, Mar. 2005.
[21] M. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Network and Distributed System Security Symposium (NDSS)*, 2012.
[22] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, 2012.
[23] J. R. Lorch, B. Parno, J. W. Mickens, M. Raykova, and J. Schiffman. Shroud: Ensuring private access to large-scale data in the data center. *FAST*, 2013:199–213, 2013.
[24] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. Phantom: Practical oblivious computation in a secure processor. ACM CCS, 2013.
[25] R. Ostrovsky. Efficient computation on oblivious rams. In *STOC*, 1990.
[26] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *STOC*, pages 294–303, 1997.
[27] B. Pinkas and T. Reinman. Oblivious RAM revisited. In *CRYPTO*, 2010.
[28] L. Ren, C. Fletcher, X. Yu, M. van Dijk, and S. Devadas. Integrity verification for path oblivious-ram. In *Proceedings of the 17th IEEE High Performance Extreme Computing Conference*, September 2013.
[29] L. Ren, X. Yu, C. Fletcher, M. van Dijk, and S. Devadas. Design space exploration and optimization of path oblivious ram in secure processors. In *Proceedings of the Int'l Symposium on Computer Architecture*, pages 571–582, June 2013. Available at Cryptology ePrint Archive, Report 2012/76.
[30] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.
[31] S. W. Smith and D. Safford. Practical server privacy with secure coprocessors. *IBM Syst. J.*, 40(3):683–695, Mar. 2001.
[32] E. Stefanov and E. Shi. Path o-ram: An extremely simple oblivious ram protocol. *CoRR*, abs/1202.5150, 2012.
[33] E. Stefanov and E. Shi. ObliviStore: High performance oblivious cloud storage. In *IEEE Symposium on Security and Privacy*, 2013.
[34] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *NDSS*, 2012.
[35] P. Williams and R. Sion. Usable PIR. In *NDSS*, 2008.
[36] P. Williams and R. Sion. Round-optimal access privacy on outsourced storage. In *CCS*, 2012.
[37] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *CCS*, 2008.
[38] P. Williams, R. Sion, and A. Tomescu. Privatefs: A parallel oblivious file system. In *CCS*, 2012.

[5] Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.