Path ORAM: An Extremely Simple Oblivious RAM Protocol

Emil Stefanov[†], Marten van Dijk[‡], Elaine Shi[*], T-H. Hubert Chan[**], Christopher Fletcher[°], Ling Ren[°], Xiangyao Yu[°], Srinivas Devadas[°]

[†]UC Berkeley [‡]UConn [*]UMD [**]University of Hong Kong [°]MIT CSAIL

Our algorithm first appeared on February 23, 2012 on arXiv.org.
http://arxiv.org/abs/1202.5150v1

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 known to date with small client storage. We formally prove that Path ORAM has a $O(\log N)$ bandwidth cost for blocks of size $B = \Omega(\log^2 N)$ bits. For such block sizes, Path ORAM is asymptotically better than the best known ORAM schemes with small client storage. Due to its practicality, Path ORAM has been adopted in the design of secure processors since its proposal.

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 [22].

Oblivious RAM (ORAM) algorithms, first proposed by Goldreich and Ostrovsky [14], 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, but also practical [4, 7, 13, 15–18, 23, 24, 26–28, 31, 34–39].

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

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

ORAM Scheme	Client Storage (# blocks of size B)	Read & Write Bandwidth (# blocks of size B)
Kushilevitz *et al.* [23] ($B = \Omega(\log N)$)	$O(1)$	$O(\log^2 N / \log\log N)$
Gentry *et al.* [12] ($B = \Omega(\log N)$)	$O(\log^2 N) \cdot \omega(1)$	$O(\log^3 N / \log\log N) \cdot \omega(1)$
Chung *et al.* [5] ($B = \Omega(\log N)$) (concurrent work)	$O(\log^{2+\epsilon}(N))$	$O(\log^2 N \cdot \log\log N) \cdot \omega(1)$
Recursive Path ORAM for small blocks, $B = \Omega(\log N)$	$O(\log N) \cdot \omega(1)$	$O(\log^2 N)$
Recursive Path ORAM for moderately sized blocks, $B = \Omega(\log^2 N)$ (we use a block size of $\Theta(\log N)$ during recursion)	$O(\log N) \cdot \omega(1)$	$O(\log N)$

Table 1: Comparison to other ORAM schemes. N is the total number of blocks and B is the block size (in bits). The failure probability is set to $N^{-\omega(1)}$ in this table, i.e., negligible in N. This table assumes that the stashes are stored on the client side.

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, 13–18, 23, 26–28, 36–38]. 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 $B = \Omega(\log^2 N)$ bits where N is the total number of blocks (e.g., 4KB blocks), recursive Path ORAM achieves an asymptotic *bandwidth cost* of $O(\log N)$ blocks, and consumes $O(\log N)\omega(1)$ blocks of client-side storage[2]. In other words, to access a single logical block, the client needs to access $O(\log N)$ 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 [23], 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 [33] in February 2012, it 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 [30] and the Ascend processor architecture [10, 11] uses Path ORAM as a primitive. Maas *et al.* [25] implemented Path ORAM on a secure processor using FPGAs and the Convey platform.

[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 $g(n) = O(f(n)h(n))$. Unless otherwise stated, all logarithms are assumed to be in base 2.

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.* [12] and Chung *et al.* [5,6]. These two works also try to improve ORAM bounds based on the binary tree construction by Shi *et al.* [31]; however, as pointed out in Section 1.1 our bound is asymptotically better than those by Gentry *et al.* [12] and Chung *et al.* [5,6]. Gentry's Path ORAM variant construction has also been applied to secure multiparty computation [12].

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 an abstract infinite ORAM construction used only for analyzing the stash usage of a non-recursive Path ORAM. For a non-recursive Path ORAM, we show that during a particular operation, the probability that the stash stores more than R blocks is at most $\leq 14 \cdot 0.6002^{-R}$. For certain choices of parameters (including R) and N data blocks, our recursive Path ORAM construction has at most $\log N$ levels, the stash at the client storage has a capacity of $R \log N$ blocks, the server storage is $20N$ blocks, and the bandwidth is $10(\log N)^2$ blocks per load/store operation. For s load/store operations, a simple union bound can show that this recursive Path ORAM fails during one of the s load/store operations (due to exceeding stash capacity) with probability at most $\leq 14s \log N \cdot 0.625^{-R}$. Choosing $R = \Theta(\log s + \log \log N) \cdot \omega(1)$ can make the failure probability negligible. We shall refine the parameters and do a more careful analysis to achieve $\Theta(\log N) \cdot \omega(1)$ blocks of client storage in the recursive construction. Our empirical results in Section 7 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 [13, 14, 26] 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, 12–16, 18, 23, 26–28, 36, 38] devoted to improving ORAM constructions. Path ORAM is based upon the binary-tree ORAM framework proposed by Shi *et al.* [31].

Near 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 [23].

Path ORAM achieves an asymptotic improvement under reasonable assumptions about the block size: when the block size is at least $\Omega(\log^2 N)$ bits, and using a smaller block size of $\Theta(\log N)$ for the position map levels, Path ORAM has bandwidth cost of $O(\log N)$. Such a block size $\Omega(\log^2 N)$ bits is what one typically encounters in most practical applications (e.g., 4KB blocks in file systems). Goldreich and Ostrovsky show that under $O(1)$ client storage, any ORAM algorithm must have bandwidth cost $\Omega(\log N)$ (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 [14, 15, 23]. Our bound partially addresses this open question for reasonably large block sizes.

Comparison with Gentry *et al.* and Chung *et al.*. Gentry *et al.* [12] improve on the binary tree ORAM scheme proposed by Shi *et al.* [31]. To achieve $2^{-\lambda}$ failure probability, their scheme achieves $O(\lambda(\log N)^2/(\log \lambda))$ blocks bandwidth cost, for block size $B = \log N$ bits. Assuming that $N = poly(\lambda)$, their bandwidth cost is $O(\lambda \log N)$ blocks. In comparison, recursive Path ORAM achieves $O(\log^2 N)$ blocks bandwidth cost when $B = \log N$. 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.* [12]. As mentioned earlier, our bound is asymptotically better than Gentry *et al.* [12] or Chung and Pass [6].

In recent concurrent and independent work, Chung *et al.* proposed another statistically secure binary-tree ORAM algorithm [5] based on Path ORAM. Their theoretical bandwidth bound is a $\log \log n$ factor worse than ours for blocks of size $\Omega(\log N)$. 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, 12, 31] (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 data 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, the data access patterns from two sequences of read/write operations with the same length must be indistinguishable.

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 [35]. 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 the sense 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 6.4. We do not focus on integrity in our main presentation.

3 The Path ORAM Protocol

We first describe the Path ORAM protocol with linear amount of client storage, and then later in Section 4 we explain how the client storage can be reduced to (poly-)logarithmic 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 and 2^L leaves. In our theoretic bounds, we need $L = \lceil \log_2(N) \rceil$, but in our experiments, we observe that $L = \lceil \log_2(N) \rceil - 1$ is sufficient. 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 leaves.

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 7.1).

N	Total # blocks outsourced to server
L	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
$x :=$ position[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

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, a small number of blocks might overflow from the tree buckets on the server. The client locally stores these overflowing blocks in a local data structure S called the stash. In Section 5, we prove that the stash has a worst-case size of $O(\log N) \cdot \omega(1)$ blocks with high probability. In fact, in Section 7.2, 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 $x :=$ position[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. For now, we assume that the position map and the stash are both stored on the client side. The position map is of size $NL = N \log N$ bits, which is of size $O(N)$ blocks when the block size $B = \Omega(\log N)$. In Section 5, we prove that the stash for the basic non-recursive Path ORAM is at most $O(\log N)\omega(1)$ blocks to obtain negligible failure probability. Later in in Section 4, we explain how the recursive construction can also achieve client storage of $O(\log N) \cdot \omega(1)$ blocks as shown in Table 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

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

filled with independent random numbers between 0 and $2^L - 1$.

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:

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 $\mathcal{P}(x, \ell) = \mathcal{P}(\mathsf{position}[\mathsf{a}'], \ell)$.

Note that when the client performs Access on a block for the first time, it will not find it in the tree or stash, and should assume that the block has a default value of zero.

Subroutines. We now explain the ReadBucket and the WriteBucket subroutine. For ReadBucket(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 WriteBucket(bucket, 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) $\Pr(\mathbf{p}) = \prod_{j=1}^{M} \Pr(\mathsf{position}_j(\mathsf{a}_j)) = (\frac{1}{2^L})^M$. 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 5: 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.

4 Recursion and Parameterization

4.1 Recursion Technique

In our non-recursive scheme described in the previous section, the client must store a relatively large position map. We can leverage the same recursion idea as described in the ORAM constructions of Stefanov *et al.* [35] and Shi *et al.* [31]. to reduce the client-side storage. The idea is simple: instead

of storing the position map on the client side, we store the position map on the server side in a smaller ORAM, and recurse.

More concretely, 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. For a more detailed description of the recursion technique, we refer the readers to [31, 35].

4.2 Parameterization

We can choose the block size for the recursive ORAMs to parametrize the recursion.

Uniform Block Size. Suppose each block has size $\chi \log N$ bits, where $\chi \geq 2$. This is a reasonable assumption that has been made by Stefanov *et al.* [34, 35] and Shi *et al.* [31]. For example, a standard 4KB block consists of 32768 bits and this assumption holds for all $N \leq 2^{16382}$. In this case, the number of level of recursions is $O(\frac{\log N}{\log \chi})$. Hence, the bandwidth overhead is $O(\frac{\log^2 N}{\log \chi})$ blocks.

(i) **Separate Local Storage for Stashes from Different Levels of Recursion.** From Theorem 1 in Section 5, in order to make the failure probability for each stash to be $\frac{1}{N^{\omega(1)}}$, the capacity of each stash can be set to $\Theta(\log N) \cdot \omega(1)$ blocks. Hence, storing stashes from all levels needs $O(\frac{\log^2 N}{\log \chi}) \cdot \omega(1)$ blocks of storage.

(ii) **Common Local Storage for Stashes from All Levels of Recursion.** A common local storage can be used to store stash blocks from all levels of recursion. From Theorem 1, it follows that the number of blocks in each stash is dominated by some geometric random variable. Hence, it suffices to analyze the sum of $O(\frac{\log N}{\log \chi})$ independent geometric random variables. From the analysis in Section 4.3 (with the detailed proof given in Section 5.7), it follows that the capacity of the common storage can be set to $\Theta(\log N) \cdot \omega(1)$ to achieve failure probability $\frac{1}{N^{\omega(1)}}$.

Non-uniform Block Size. Suppose that the *regular block* size is $B = \Omega(\log^2 N)$ bits. This is the block size for the original ORAM (ORam_0). Suppose for the position map ORAMs (ORam_i for $i \geq 1$), we use a *small block* size of $\chi \log_2 N$ bits for some constant $\chi \geq 2$.

The client storage is $O(\log^2 N) \cdot \omega(1)$ small blocks and $O(\log N) \cdot \omega(1)$ regular blocks. Hence, the total client storage is $O(\log N) \cdot \omega(1)$ regular blocks. Each operation involves communication of $O(\log^2 N)$ small blocks and $O(\log N)$ regular blocks. Hence, the bandwidth cost (with respect to a regular block) is $O(\log N)$.

As can be seen, using non-uniform block sizes leads to a much more efficient parametrization of recursive Path ORAM in terms of bandwidth. However, using uniform block sizes reduces the number of recursion levels, leading to a smaller round-trip response time.

4.3 Shared Stash

In Section 5.7, we show that the client storage for our Recursive Path ORAM construction (with non-uniform block sizes) can be reduced from $O(\log^2 N) \cdot \omega(1)$ to $O(\log N) \cdot \omega(1)$ by having a *single*

stash shared among all levels of the recursion. This is possible while still maintaining a negligible probability of stash overflow.

5 Bounds on Stash Usage

In this section we will analyze the stash usage for a non-recursive Path-ORAM, where each bucket in the Path-ORAM binary tree stores a *constant* number of blocks. In particular, we analyze the probability that, after a sequence of load/store operations, the number of blocks in the the stash exceeds R, and show that this probability decreases exponentially in R.

By ORAM_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 leaves 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 $\mathbf{a} = (a_i)_{i=1}^s$ of block addresses of blocks that are loaded/stored, (2) the sequence of labels $\mathbf{x} = (x_i)_{i=1}^s$ as seen by the server (line 1 in Figure 1), and (3) the sequence $\mathbf{y} = (y_i)_{i=1}^s$ of remapped leaf labels (line 2 in Figure 1). The tuple (a_1, x_1, y_1) corresponds to the most recent load/store operation, (a_2, x_2, y_2) corresponds to the next most recent load/store operation, and so on. A path from the root to some x_i is known as an *eviction* path, and a path from the root to some y_i is known as an *assigned* path. 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_i in $\mathbf{a}$. We write $a(\mathbf{s}) = \mathbf{a}$.

By $\mathsf{ORAM}_L^Z[\mathbf{s}]$ we denote the distribution of real blocks in ORAM_L^Z after a sequence $\mathbf{s}$ of load/store operations starting with an empty ORAM; the sequence $\mathbf{s}$ 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 $\mathsf{st}(\mathsf{ORAM}_L^Z[\mathbf{s}])$ 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 $\mathsf{st}(\mathsf{ORAM}_L^Z[\mathbf{s}]) > R$.

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* $Z = 5$*, tree height* $L = \lceil \log N \rceil$ *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*

$$\Pr(\mathsf{st}(\mathsf{ORAM}_L^5[\mathbf{s}]) > R \,|\, a(\mathbf{s}) = \mathbf{a}) \leq 14 \cdot (0.6002)^R,$$

where the probability is over the randomness that determines $\mathbf{x}$ *and* $\mathbf{y}$ *in* $\mathbf{s} = (\mathbf{a}, \mathbf{x}, \mathbf{y})$*.*

As a corollary, for s load/store operations on N data blocks, Path ORAM with client storage $\leq R$ blocks, server storage $20N$ blocks and bandwidth $10 \log N$ blocks per load/store operation, fails during one of the s load/store operations with probability $\leq s \cdot 14 \cdot 0.6002^R$. 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 the process 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 ∞-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 $\Pr[\mathsf{st}(\mathsf{ORAM}_L^Z[\mathbf{s}]) > R \,|\, a[\mathbf{s}] = \mathbf{a}\ )$ 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 measure concentration techniques we prove the main theorem.

5.1 ∞-ORAM

We define ∞-ORAM, denoted by ORAM_L^∞, 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 ∞-ORAM is used only as a tool for usage analysis, and does not need to respect any security notions.

In order to use ORAM_L^∞ to analyze the stash usage of ORAM_L^Z, we define a *post-processing* greedy algorithm G_Z that takes as input the state of ORAM_L^∞ after a sequence $\mathbf{s}$ of load/store operations and attempts to reassign blocks such that each bucket stores at most Z blocks, putting excess blocks in the (initially empty) stash if necessary. We use $\mathsf{st}^Z(\mathsf{ORAM}_L^\infty[\mathbf{s}])$ to denote the stash usage after the greedy algorithm is applied. The algorithm repeats the following steps until there is no bucket that stores more than Z blocks, where the stash is initially empty.

1. Select a block in a bucket that stores more than Z blocks. Suppose the bucket is at level h, and P is the path from the bucket to the root.

2. Find the highest level $i \leq h$ such that the bucket at level i on the path P stores less than Z blocks. If such a bucket exists, then use it to store the block. If it does not exist, then put 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:*

$$\mathsf{st}^Z(\mathsf{ORAM}_L^\infty[\mathbf{s}]) = \mathsf{st}(\mathsf{ORAM}_L^Z[\mathbf{s}]).$$

Proof. We first notice that the order in which the greedy strategy G_Z 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 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 f. After reading the path to leaf f into the cache, Path-ORAM moves blocks from its cache/stash into the path from root to leaf f according to algorithm G_Z. Therefore, post-processing 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_Z. The number of real blocks in the stash is the same and this proves the lemma. □

5.2 Usage/Capacity Bounds

To investigate when a not processed ORAM_L^∞ can lead to a stash usage of $> R$ after post-processing, we start by analyzing bucket usage over subtrees. When we talk about a subtree T of the binary tree, we always implicitly assume that it contains the root of the ORAM tree; in particular, if a node is contained in T, then so are all its ancestors. We define $n(T)$ to be the total number of nodes in T. For ∞-ORAM we define the usage $\mathsf{u}^T(\mathsf{ORAM}_L^\infty[\mathbf{s}])$ 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.

The following lemma characterizes the stash usage:

Lemma 2. *The stash usage* $\mathsf{st}^Z(\mathsf{ORAM}_L^\infty[\mathbf{s}])$ *in post-processed* ∞*-ORAM is* $> R$ *if and only if there exists a subtree* T *in* ORAM_L^∞ *such that* $\mathsf{u}^T(\mathsf{ORAM}_L^\infty[\mathbf{s}]) > n(T) \cdot Z + R$.

Proof. If part: Suppose T is a subtree such that $\mathsf{u}^T(\mathsf{ORAM}_L^\infty[\mathbf{s}]) > n(T) \cdot Z + R$. Observe that the greedy algorithm can assign the blocks in a bucket only to an ancestor bucket. Since T can store at most $n(T) \cdot Z$ blocks, more than R blocks must be assigned to the stash by the greedy algorithm G_Z.
Only if part: Suppose that $\mathsf{st}^Z(\mathsf{ORAM}_L^\infty[\mathbf{s}]) > R$. Define T to be the maximal subtree that contains all buckets with exactly Z blocks after post-processing by the greedy algorithm G_Z. Suppose b is a bucket not in T. By the maximality of T, there is an ancestor (not necessarily proper ancestor) bucket b' of b that contains less than Z blocks after post-processing, which implies that no block from b can go to the stash. Hence, all blocks that are in the stash must have originated from a bucket in T. Therefore, it follows that $\mathsf{u}^T(\mathsf{ORAM}_L^\infty[\mathbf{s}]) > n(T) \cdot Z + R$ □

Lemma 3 (Worst-case Address Pattern)**.** *Out of all address sequences* $\mathbf{a}$ *on a working set of size* N*, the probability* $\Pr[\mathsf{st}^Z(\mathsf{ORAM}_L^\infty[\mathbf{s}]) > R | a(\mathbf{s}) = \mathbf{a}]$ *is maximized by a sequence* $\mathbf{a}$ *in which each block address appears exactly once, i.e.,* $s = N$ *and there are no duplicate block addresses. As*

a consequence, for such an address pattern, the labels in $(x_i)_{i=1}^N$ and $(y_i)_{i=1}^N$ are all statistically independent of one another.

Proof. 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, the probability $\Pr[\mathsf{st}^Z(\mathsf{ORAM}_L^\infty[\mathbf{s}]) > R | a(\mathbf{s}) = \mathbf{a}]$ is maximized when $\mathbf{a} = (a_i)_{i=1}^s$ is a sequence of block addresses without duplicates. □

Bounding Usage for Each Subtree. In view of Lemma 3, we fix a sequence $\mathbf{a}$ of N distinct block addresses. The randomness comes from the independent choices of labels $(x_i)_{i=1}^N$ and $(y_i)_{i=1}^N$.

As a corollary to Lemmas 1 and 2, we obtain

$$\begin{aligned} & \Pr[\mathsf{st}(\mathsf{ORAM}_L^Z[\mathbf{s}]) > R] \\ = \; & \Pr[\mathsf{st}^Z(\mathsf{ORAM}_L^\infty[\mathbf{s}]) > R] \\ = \; & \Pr[\exists T \; \mathsf{u}^T(\mathsf{ORAM}_L^\infty[\mathbf{s}]) > n(T)Z + R] \\ \leq \; & \sum_T \Pr[\mathsf{u}^T(\mathsf{ORAM}_L^\infty[\mathbf{s}]) > n(T)Z + R], \end{aligned}$$

where T ranges over all subtrees containing the root, and the inequality follows from the union bound.

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

$$\begin{gathered} \Pr[\mathsf{st}(\mathsf{ORAM}_L^Z[\mathbf{s}]) > R] \\ \leq \textstyle\sum_{n \geq 1} 4^n \max_{T:n(T)=n} \Pr[\mathsf{u}^T(\mathsf{ORAM}_L^\infty[\mathbf{s}]) > nZ + R]. \end{gathered}$$

We next give a uniform upper bound for

$$\Pr[\mathsf{u}^T(\mathsf{ORAM}_L^\infty[\mathbf{s}]) > nZ + R]$$

in terms of n, Z and R.

5.3 Usage of Subtree via Eviction Game

In view of Lemma 3, we assume that the request sequence is of length N, and consists of distinct block addresses. Recall that the $2N$ labels in $\mathbf{x} = (x_i)_{i=1}^N$ and $\mathbf{y} = (y_i)_{i=1}^N$ are independent, where each label corresponds to a leaf bucket. The indices i are given in reversed order, i.e, $i = N$ is the first access and $i = 1$ is the last access.

At every time step i, a block is requested. The block resides in a random path with some label x_i previously chosen (however the random choice has never been revealed). This path is now visited. Henceforth, this path is called the *eviction path* $P_{\text{evict}}(i)$.

Then, the block is logically assigned to a freshly chosen random path with label y_i, referred to as the *assigned path* $P_{\text{assign}}(i)$.

Variable	Meaning
T	a subtree rooted at the root of the ORAM_∞ binary tree
$\widehat{T}$	augmented tree by including every child (if any) of every node in T
F	nodes of a subtree T that are leaves to the ORAM binary tree
E	set of exit nodes of a subtree T
$\widehat{E} := E \cup F$	set of all leaves of $\widehat{T}$
Z	capacity of each bucket

Table 3: Notations for the analysis.

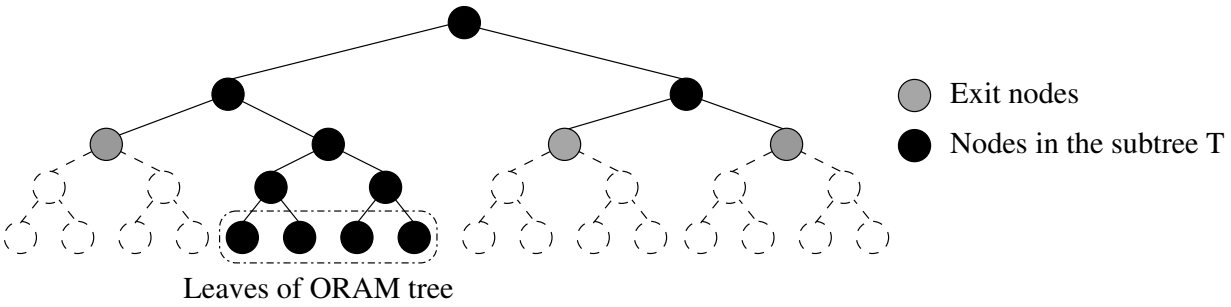


Figure 2: A subtree containing some leaves of the original ORAM binary tree, augmented with the exit nodes

Recall that given subtree T, we want to probabilistically bound the quantity $u^T(\mathsf{ORAM}_L^\infty[\mathbf{s}])$, which is the number of blocks that, at the end of an access sequence $\mathbf{s}$, survive in the subtree T, i.e., have not been evicted out of the subtree T.

Definition 2 (Exit node)**.** *For a given path P leading to some leaf node, suppose that some node u is the first node of the path P that is not part of T, then we refer to node u as the exit node, denoted $u := \mathsf{exit}(P, T)$. If the whole path P is contained in T, then the exit node $\mathsf{exit}(P, T)$ is* null*.*

To bound the number of surviving blocks in T at the end of an access sequence of N, we focus on an individual block that is requested at step i in this sequence. Suppose this block resides in path $P_{\text{evict}}(i)$, and is assigned to some path $P_{\text{assign}}(i)$ at the end of the request. Observe that this block will be evicted from tree T at the end of the sequence *iff* the following holds: there exists a later step $j \le i$ (recalling that smaller indices mean later) such that both exit nodes $\mathsf{exit}(P_{\text{evict}}(j), T)$ and $\mathsf{exit}(P_{\text{assign}}(i), T)$ are equal and not null. Otherwise the block will survive in tree T at the end of the sequence — and will contribute to the count of the total surviving blocks.

We next define some notations relating to the subtree T.

Let F be the set of nodes in T that are also leaves of the ORAM binary tree; we denote $l := |F|$. We augment the tree T by adding nodes to form $\widehat{T}$ in the following way. If a node in T has any child node v that is not in T, then node v will be added to $\widehat{T}$. The added nodes in $\widehat{T}$ are referred to as exit nodes, denoted by E; the leaves of $\widehat{T}$ are denoted by $\widehat{E} = E \cup F$. Observe that if T contains n nodes and $|F| = l$, then $|\widehat{E}| = n - l + 1$.

We summarize the notations we use in Table 3.

Eviction Game. Consider the following eviction game, which is essentially playing the ORAM access sequence *reversed in time*. Initially, all nodes in $\widehat{E}$ are marked *closed*; all nodes in F will

remain closed, and only nodes in E may be marked *open* later.

For $i = 1, 2, \ldots, N$, do the following:

1. Pick a random *eviction path* starting from the root, denoted $P_{\text{evict}}(i)$. The path will intersect exactly one node v in $\widehat{E}$. If v is an exit node in E, mark node v as open (if that exit node is already marked open, then it will continue to be open).

2. For a block requested in time step i, pick a random assignment path $P_{\text{assign}}(i)$ starting from the root. The path $P_{\text{assign}}(i)$ will intersect exactly one node in $\widehat{E}$. If this node is closed, then the block *survives*; otherwise, the block is *evicted* from T.

For $i \in [N]$, a block b_i from step i survives because its corresponding assignment path intersects a node in F, or an exit node that is closed. We define X_i to be the indicator variable that equals 1 *iff* the path $P_{\text{assign}}(i)$ intersects a node in F (and block b_i survives), Y_i to be the indicator variable that equals 1 *iff* the path $P_{\text{assign}}(i)$ intersects a node in E and block b_i survives. Define $X := \sum_{i\in[N]} X_i$ and $Y := \sum_{i\in[N]} Y_i$; observe that the number of blocks that survive is $X + Y$.

5.4 Negative Association

Independence is often required to show measure concentration. However, the random variables X and Y are not independent. Fortunately, X and Y are negatively associated. For simplicity, we show a special case in the following lemma, which will be useful in the later measure concentration argument.

Lemma 4. *For X and Y defined above, and all $t \geq 0$, $E[e^{t(X+Y)}] \leq E[e^{tX}] \cdot E[e^{tY}]$.*

Proof. Observe that there exists some $p \in [0, 1]$, such that for all $i \in [N]$, $p = \Pr[X_i = 1]$. For each $i \in [N]$, define $q_i := \Pr[Y_i = 1]$. Observe that the q_i's are random variables, and they are non-increasing in i and determined by the choice of eviction paths.

For each $i \in [N]$, $X_i + Y_i \leq 1$, and hence X_i and Y_i are negatively associated. Conditioning on $\mathbf{q} := (q_i)_{i\in[N]}$, observe that the (X_i, Y_i)'s are determined by the choice of independent assignment paths in different rounds, and hence are independent over different i's. Hence, it follows from [9, Proposition 7(1)] that conditioning on $\mathbf{q}$, X and Y are negatively associated.

In particular, for non-negative t, we have $E[e^{t(X+Y)}|\mathbf{q}] \leq E[e^{tX}|\mathbf{q}] \cdot E[e^{tY}|\mathbf{q}] = E[e^{tX}] \cdot E[e^{tY}|\mathbf{q}]$, where the last equality follows because X is independent of $\mathbf{q}$. Taking expectation over $\mathbf{q}$ gives $E[e^{t(X+Y)}] \leq E[e^{tX}] \cdot E[e^{tY}]$. □

5.5 Stochastic Dominance

Because of negative association, we consider two games to analyze the random variables X and Y separately. The first game is a balls-and-bins game which produces a random variable $\widehat{X}$ that has the same distribution as X. The second game is a modified eviction game with countably infinite number of rounds, which produces a random variable $\widehat{Y}$ that stochastically dominates Y, in the sense that $\widehat{Y}$ and Y can be coupled such that $Y \leq \widehat{Y}$.

1. **Balls-and-Bins Game:** For simplicity, we assume $N = 2^L$. In this game, N blocks are thrown independently and uniformly at random into N buckets corresponding to the leaves of the ORAM binary tree. The blocks that fall in the leaves in F survive. Observe that the number $\widehat{X}$ of surviving blocks has the same distribution as X.

2. **Infinite Eviction Game:** This is the same as before, except for the following differences.

   (a) Every node in $\widehat{E}$ is initially closed, but all of them could be open later. In particular, if some eviction path $P_{\text{evict}}(i)$ intersects a leaf node $v \in F$, node v will become open.

   (b) There could be countably infinite number of rounds, until eventually all nodes in $\widehat{E}$ are open, and no more blocks can survive after that.

   Let $\widehat{Y}$ be the total number of surviving blocks in the infinite eviction game. Observe that a block with assigned path intersecting a node in F will not be counted towards Y, but might be counted towards $\widehat{Y}$ if the node is closed. Hence, there is a natural coupling such that the number of surviving blocks in the first N rounds in the infinite game is at least Y in the finite eviction game. Hence, the random variable $\widehat{Y}$ stochastically dominates Y. Hence, we have for all non-negative t, $E[e^{tY}] \leq E[e^{t\widehat{Y}}]$.

5.6 Measure Concentration for the Number of Surviving Blocks in Subtree

We shall find parameters Z and R such that, with high probability, $X + Y$ is at most $nZ + R$. For simplicity, we assume $N = 2^L$ is a power of two.

Lemma 5. *Suppose that the address sequence is of length N, where $L := \lceil \log_2 N \rceil$. Moreover, suppose that T is a subtree of the binary ORAM tree containing the root having $n = n(T)$ nodes. Then, for $Z = 5$, for any $R > 0$, $\Pr[\mathsf{u}^T(\mathsf{ORAM}_L^\infty[\mathbf{s}]) > n \cdot Z + R] \leq \frac{1}{4^n} \cdot (0.9332)^n \cdot e^{-0.5105R}$.*

Proof. Because of negative association between X and Y, and stochastic dominance by $\widehat{X}$ and $\widehat{Y}$, we analyze $\widehat{X}$ and $\widehat{Y}$.

Observe that if T has n nodes, among which l are in F, then $\widehat{T}$ has $n - l + 1$ leaves in $\widehat{E}$.

Balls-and-Bins Game. Recall that X_i is the indicator variable for whether the assigned path $P_{\text{assign}}(i)$ intersects a bucket in F. Then, $\widehat{X} = \sum_{i \in [n]} X_i$. Recall that $l = |F|$ and there are $N = 2^L$ leaf buckets in the binary ORAM tree. Observe that for real t, $E[e^{tX_i}] = (1 - \frac{l}{N}) + \frac{l}{N} e^t \leq \exp(\frac{l}{N}(e^t - 1))$, and hence by independence, $E[e^{t\widehat{X}}] \leq \exp(l(e^t - 1))$.

Infinite Eviction Game. For each $v \in \widehat{E}$, suppose v is at depth d_v (the root is at depth 0), and let its weight be $w(v) := \frac{1}{2^{d_v}}$. Observe that the sum of weights of nodes in $\widehat{E}$ is 1.

Define M_j to be the number of surviving blocks such that there are exactly j open nodes at the moment when the corresponding assigned paths are chosen. Since $\widehat{E}$ contains $n - l + 1$ nodes, we consider j from 1 to $k = n - l$. (Observe that for in the first round of the infinite eviction game, the first eviction path closes one of the nodes in $\widehat{E}$, and hence, the number of remaining open nodes for the first block is $n - l$.) Let Q_j be the number of rounds in which, at the moment when the assigned path is chosen, there are exactly j open nodes. Let w_j be the weight of the jth open node. Define $q_j := 1 - \sum_{i=1}^{j} w_i$. Observe that conditioning on q_j, Q_j follows the geometric distribution with parameter q_j; moreover, conditioning on q_j and Q_j, M_j is the sum of Q_j independent Bernoulli random variables, each with expectation q_j.

Define $\widehat{Y} := \sum_{j=1}^{k} M_j$. We shall analyze the moment generating function $E[e^{t\widehat{Y}}]$ for appropriate values of $t > 0$. The strategy is that we first derive an upper bound $\phi(t)$ for $E[e^{tM_k}|\mathbf{w}]$, where $\mathbf{w} := \{w_j\}_{j \leq k}$, that depends only on t (and in particular independent of $\mathbf{w}$ or k). This allows us to conclude that $E[e^{t\widehat{Y}}] \leq (\phi(t))^k$.

For simplicity, in the below argument, we assume that we have conditioned on $\mathbf{w} = \{w_j\}_{j \le k}$ and we write $Q = Q_k$, $M = M_k$ and $q = q_k$.

Recall that M is a sum of Q independent Bernoulli random variables, where Q has a geometric distribution. Therefore, we have the following.

$$\begin{aligned}
E[e^{tM}|\mathbf{w}] &= \sum_{i \ge 1} q(1-q)^i E[e^{tM}|Q=i,\mathbf{w}] && (1)\\
&\le \sum_{i \ge 1} q(1-q)^{i-1} \exp(qi(e^t-1)) && (2)\\
&= \frac{q\exp(q(e^t-1))}{1-(1-q)\exp(q(e^t-1))} && (3)\\
&= \frac{q}{\exp(-q(e^t-1))-(1-q)} && (4)\\
&\le \frac{q}{1-q(e^t-1)-1+q} && (5)\\
&= \frac{1}{2-e^t}. && (6)
\end{aligned}$$

From (1) to (2), we consider the moment generating function of a sum of i independent Bernoulli random variables, each having expectation q: $E[e^{tM}|Q=i,\mathbf{w}] = ((1-q)+qe^t)^i \le \exp(qi(e^t-1))$. In (3), for the series to converge, we observe that $(1-q)\exp(q(e^t-1)) \le \exp(q(e^t-2))$, which is smaller than 1 when $0 < t < \ln 2$. In (5), we use $1-u \le \exp(-u)$ for all real u.

Hence, we have shown that for $0 < t < \ln 2$, $E[e^{t\sum_{j=1}^{k} M_j}|\mathbf{w}] = E[e^{t\sum_{j=1}^{k-1} M_j}|\mathbf{w}] \cdot E[e^{tM_k}|\mathbf{w}]$ (since conditioning on $\mathbf{w}$, M_k is independent of past history), which we show from above is at most $E[e^{t\sum_{j=1}^{k-1} M_j}|\mathbf{w}] \cdot (\frac{1}{2-e^t})$. Taking expectation over $\mathbf{w}$ gives $E[e^{t\sum_{j=1}^{k} M_j}] \le E[e^{t\sum_{j=1}^{k-1} M_j}] \cdot (\frac{1}{2-e^t})$.

Observe that the inequality holds independent of the value of $q = q_k$. Therefore, the same argument can be used to prove that, for any $1 < \kappa \le k$, we have $E[e^{t\sum_{j=1}^{\kappa} M_j}] \le E[e^{t\sum_{j=1}^{\kappa-1} M_j}] \cdot (\frac{1}{2-e^t})$.

Hence, a simple induction argument on $k = n - l$ can show that $E[e^{t\widehat{Y}}] \le (\frac{1}{2-e^t})^{n-l}$.

Combining Together. Let Z be the capacity for each bucket in the binary ORAM tree, and R be the number blocks overflowing from the binary tree that need to be stored in the stash.

We next perform a standard moment generating function argument. For $0 < t < \ln 2$, $\Pr[X+Y \ge nZ+R] = \Pr[e^{t(X+Y)} \ge e^{t(nZ+R)}]$, which by Markov's Inequality, is at most $E[e^{t(X+Y)}] \cdot e^{-t(nZ+R)}$, which, by negative associativity and stochastic dominance, is at most $E[e^{t\widehat{X}}] \cdot E[e^{t\widehat{Y}}] \cdot e^{-t(nZ+R)} \le (\frac{e^{-tZ}}{2-e^t})^{n-l} \cdot e^{l(e^t-1-tZ)} \cdot e^{-tR}$.

Putting $Z = 5$, and $t = 0.5105$, one can check that $\max\{\frac{e^{-tZ}}{2-e^t}, e^{e^t-1-tZ}\} \le \frac{1}{4} \cdot 0.9332$, and so the probability is at most $\frac{1}{4^n} \cdot (0.9332)^n \cdot e^{-0.5105R}$. □

Proof of Theorem 1. By applying Lemma 5 to inequality (1), we have the following: $\Pr[\mathsf{st}(\mathsf{ORAM}_L^Z[\mathbf{s}]) > R|a(\mathbf{s}) = \mathbf{a}] \le \sum_{n \ge 1} 4^n \cdot \frac{1}{4^n} \cdot (0.9332)^n \cdot e^{-0.5105R} \le 14 \cdot (0.6002)^R$, as required.

5.7 Bounds for Shared Stash

We now show that if all levels of the recursion use the same stash, the stash size is $O(\log N) \cdot \omega(1)$ with high probability.

Suppose there are $K = O(\log N)$ levels of recursion in the recursive Path ORAM. We consider a moment after a sequence of ORAM operations are executed. For $k \in [K]$, let S_k be the number of blocks in the stash from level k. From Theorem 1, for each $k \in [K]$, for each $R > 0$, $\Pr[S_k > R] \leq 14 \cdot (0.6002)^R$. Observing that a geometric distribution G with parameter p satisfies $\Pr[G > R] \leq (1-p)^R$, we have the following.

Proposition 1. *For each $k \in [K]$, the random variable S_k is stochastically dominated by $3 + G$, where G is the geometric distribution with parameter $p = 1 - 0.6002 = 0.3998$.*

From Proposition 1, it follows that the number of stash blocks in the common storage is stochastically dominated by $3K + \sum_{k\in[K]} G_k$, where G_k's are independent geometric distribution with parameter $p = 0.3998$. It suffices to perform a standard measure concentration analysis on a sum of independent geometrically distributed random variables, but we need to consider the case that the sum deviates significantly from its mean, because we want to achieve negligible failure probability.

Lemma 6 (Sum of Independent Geometric Distributions)**.** *Suppose $(G_k : k \in [K])$ are independent geometrically distributed random variables with parameter $p \in (0,1)$. Then, for $R > 0$, we have $\Pr[\sum_{k\in[K]} G_k > E[\sum_{k\in[K]} G_k] + R] \leq \exp(-\frac{pR}{2} + \frac{K}{2})$.*

Proof. We use the standard method of moment generating function. For $t \leq \frac{p}{2}$, we have, for each $k \in [K]$,

$$\begin{aligned}
E[e^{tG_k}] &= \sum_{i\geq 1} p(1-p)^i \cdot e^{it} && (7)\\
&= \frac{pe^t}{1-(1-p)e^t} = \frac{p}{p+e^{-t}-1} && (8)\\
&\leq \frac{p}{p-t} = \frac{1}{1-\frac{t}{p}} && (9)\\
&\leq 1 + \frac{t}{p} + \frac{2t^2}{p^2} && (10)\\
&\leq \exp(\frac{t}{p} + \frac{2t^2}{p^2}), && (11)
\end{aligned}$$

where in (8), the geometric series converges, because $t \leq \frac{p}{2} < \ln \frac{1}{1-p}$, for $0 < p < 1$. In (9), we use the inequality $1 - e^{-t} \leq t$; in (10), we use the inequality $\frac{1}{1-u} \leq 1 + u + 2u^2$, for $u \leq \frac{1}{2}$.

Observing that $E[G_k] = \frac{1}{p}$, we have for $0 < t \leq \frac{p}{2}$,

$\Pr[\sum_{k\in[K]} G_k > E[\sum_{k\in[K]} G_k] + R] \leq E[\exp(t\sum_{k\in[K]} G_k)] \cdot \exp(-t(\frac{K}{p} + R)) \leq \exp(-tR + \frac{2t^2K}{p^2})$. Putting $t = \frac{p}{2}$, we have $\Pr[\sum_{k\in[K]} G_k > E[\sum_{k\in[K]} G_k] + R] \leq \exp(-\frac{pR}{2} + \frac{K}{2})$, as required. □

From Lemma 6, observing that $K = O(\log N)$ and $p = 0.3998$, to achieve failure probability $\frac{1}{N^{\omega(1)}}$, it suffices to set the capacity of the common stash storage to be $\frac{1}{p} \cdot (O(K) + (\log N) \cdot \omega(1)) = \Theta(\log N) \cdot \omega(1)$ blocks.

6 Applications and Extensions

6.1 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 [12]. 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 $O(\log^2 N)$ 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 $O((\log N)^3 / \log\log N)$.

6.2 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 [19] 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 $O(\log N) \cdot \omega(1)$ and the bandwidth is $O(\log N)$ when $B = \Omega(\log^2 N)$, we can reduce the permanent client state to $O(1)$ and achieve a bandwidth of $O(\log N) \cdot \omega(1)$. Note that *during* an access the client still needs about $O(\log N) \cdot \omega(1)$ *transient* client storage to perform the Access operation, but after the Access operation completes, the client only needs to store the private key.

For smaller blocks when $B = \Omega(\log N)$, we can achieve $O(1)$ permanent client storage, $O(\log N) \cdot \omega(1)$ transient client storage, and $O(\log^2 N)$ bandwidth cost.

6.3 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.* [10, 11] and Ren *et al.* [30] 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.* [25] built a hardware implementation of a Path ORAM based secure processor using FPGAs and the Convey platform.

Ren *et al.* [30] and Maas *et al.* [25] 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.

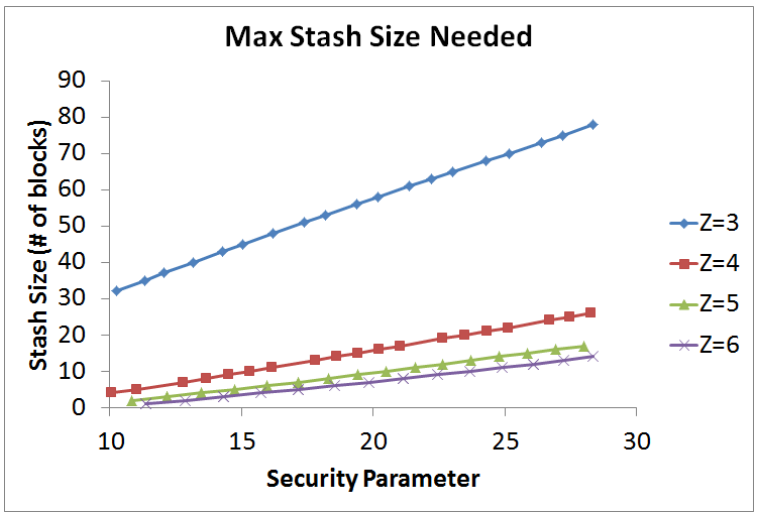


Figure 3: Empirical estimation of the required stash size to achieve failure probability less than $2^{-\lambda}$ where λ is the security parameter. Measured for $N = 2^{16}$, but as Figure 4 shows, the stash size does not depend on N (at least for $Z = 4$). The measurements represent a worst-case (in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

6.4 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) need to be read or written for each ReadBucket or WriteBucket operation.

In [29], Ren *et al.* further optimize the integrity verification overhead for the recursive Path ORAM construction.

7 Evaluation

In our experiments, the Path ORAM uses a binary tree with height $L = \lceil \log_2(N) \rceil - 1$.

7.1 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*

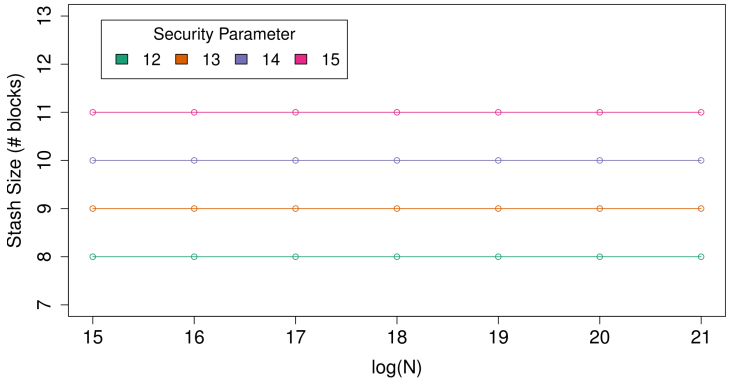


Figure 4: The stash size to achieve failure probability less than $2^{-\lambda}$ does not depend on N ($Z = 4$). Measured for a worst-case (in terms of stash size) access pattern. The stash size does not include the temporarily fetched path during Access.

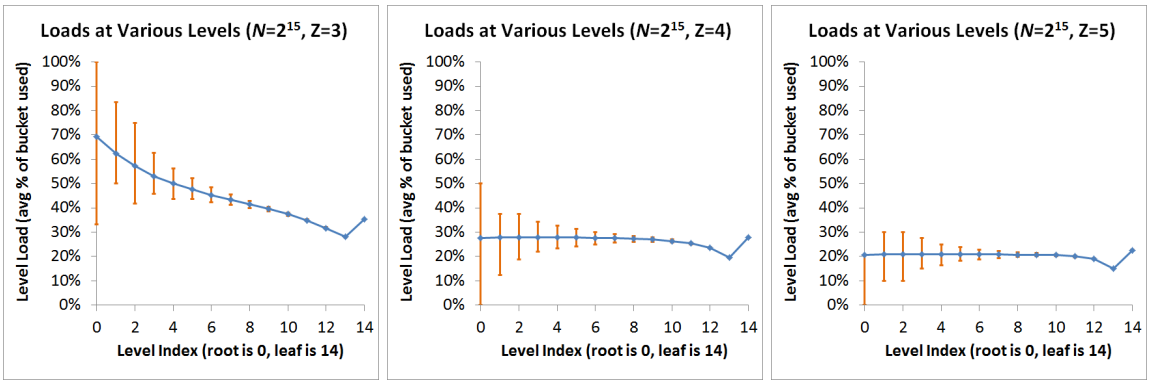
Security Parameter (λ)	**Bucket Size (Z)**		
	4	**5**	**6**
	Max Stash Size		
80	89	63	53
128	147	105	89
256	303	218	186

Figure 5: 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.

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 5 shows the probability of exceeding stash capacity 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 4 and Figure 3. 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 $\{1, 2, \ldots, N, 1, 2, \ldots, N, 1, 2, \ldots\}$. In section 5, we show that 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 [20]).

Figure 3 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 Figure 5, we extrapolate those results for realistic values of λ. The experiments show that the required stash size grows linearly with the security parameter,



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

Figure 6: 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.

which is in accordance with the Main Theorem in Section 5 that the failure probability decreases exponentially with the stash size. Figure 4 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 $Z \geq 5$, in practice, the stash capacity is not exceeded with high probability when $Z = 4$. $Z = 3$ behaves relatively worse in terms of stash occupancy, and it is unclear how likely the stash capacity is exceeded when $Z = 3$.

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.* [25].

7.2 Bucket Load

Figure 6 gives the bucket load per level for $Z \in \{3, 4, 5\}$. We prove in Section 5 that for $Z \geq 5$, the expected usage of a subtree T is close to the number of buckets in it. And Figure 6 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 7.1 and shown in Figure 3, $Z = 3$ produces much larger stash sizes in practice).

8 Conclusion

Partly due to its simplicity, Path ORAM is the most practical ORAM scheme known-to-date under a small amount of client storage. We formally prove asymptotic bounds on Path ORAM, and show that its performance is competitive with or asymptotically better than the the best known

construction (for a small amount of client storage), assuming reasonably large block sizes. We also present simulation results that confirm our theoretic bounds.

Acknowledgments

Emil Stefanov was supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-0946797 and by a DoD National Defense Science and Engineering Graduate Fellowship. Elaine Shi is supported by NSF under grant CNS-1314857. Christopher Fletcher was supported by a National Science Foundation Graduate Research Fellowship, Grant No. 1122374 and by a DoD National Defense Science and Engineering Graduate Fellowship. This research was partially supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract N66001-10-2-4089, and a grant from the Amazon Web Services in Education program. 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.

We would like to thank Kai-Min Chung and Jonathan Katz for helpful discussions. We would like to thank Kai-Min Chung for pointing out that our algorithm is information-theoretically (statistically) secure.

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] D. P. Dubhashi and D. Ranjan. Balls and bins: A study in negative dependence. *Random Struct. Algorithms*, 13(2):99–124, 1998.

[10] 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.

[11] C. W. Fletcher. Ascend: An architecture for performing secure computation on encrypted data. In *MIT CSAIL CSG Technical Memo 508*, April 2013.

[12] 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.

[13] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *STOC*, 1987.

[14] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 1996.

[15] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, 2011.

[16] 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.

[17] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Practical oblivious storage. In *CODASPY*, 2012.

[18] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, 2012.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, 2012.

[24] 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.

[25] 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.

[26] R. Ostrovsky. Efficient computation on oblivious rams. In *STOC*, 1990.

[27] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *STOC*, pages 294–303, 1997.

[28] B. Pinkas and T. Reinman. Oblivious RAM revisited. In *CRYPTO*, 2010.

[29] 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.

[30] 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.

[31] 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.

[32] S. W. Smith and D. Safford. Practical server privacy with secure coprocessors. *IBM Syst. J.*, 40(3):683–695, Mar. 2001.

[33] E. Stefanov and E. Shi. Path o-ram: An extremely simple oblivious ram protocol. *CoRR*, abs/1202.5150, 2012.

[34] E. Stefanov and E. Shi. ObliviStore: High performance oblivious cloud storage. In *IEEE Symposium on Security and Privacy*, 2013.

[35] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *NDSS*, 2012.

[36] P. Williams and R. Sion. Usable PIR. In *NDSS*, 2008.

[37] P. Williams and R. Sion. Round-optimal access privacy on outsourced storage. In *CCS*, 2012.

[38] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *CCS*, 2008.

[39] P. Williams, R. Sion, and A. Tomescu. Privatefs: A parallel oblivious file system. In *CCS*, 2012.