Path ORAM: An Extremely Simple Oblivious RAM Protocol

EMIL STEFANOV, UC Berkeley
MARTEN VAN DIJK, University of Connecticut
ELAINE SHI, Cornell University
T.-H. HUBERT CHAN, University of Hong Kong
CHRISTOPHER FLETCHER, University of Illinois at Urbana-Champaign
LING REN, XIANGYAO YU, and SRINIVAS DEVADAS, MIT CSAIL

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.

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

General Terms: Algorithms, Security

Additional Key Words and Phrases: Oblivious RAM, ORAM, Path ORAM, access pattern

ACM Reference format:
Emil Stefanov, Marten Van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2018. Path ORAM: An Extremely Simple Oblivious RAM Protocol. *J. ACM* 65, 4, Article 18 (April 2018), 26 pages.
https://doi.org/10.1145/3177872

A conference version of the article has appeared in ACM Conference on Computer and Communications Security (CCS), 2013.
This work is partially supported by the NSF Graduate Research Fellowship grants DGE-0946797 and DGE-1122374, the DoD NDSEG Fellowship, NSF grant CNS-1314857, DARPA CRASH program 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.
The research was supported in part by a grant from Hong Kong RGC under the contract HKU719312E.
Authors' addresses: E. Stefanov, Department of Electrical Engineering and Computer Sciences, UC Berkeley, CA 94720, USA; email: emil@berkeley.edu; M. V. Dijk, Electrical and Computing Engineering Department, University of Connecticut, Storrs-Mansfield, CT 06269, USA; email: vandijk@engr.uconn.edu; E. Shi, Department of Computer Science, Cornell University, Ithaca, NY 14853-7501, USA; email: elaine@cs.cornell.edu; T.-H. H. Chan, Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong; email: hubert@cs.hku.hk; C. Fletcher, Computer Science Department, University of Illinois–Urbana Champaign, Urbana, IL 61801, USA; email: cwfletch@illinois.edu; L. Ren, X. Yu, and S. Devadas, MIT CSAIL, Cambridge, MA 02139, USA; emails: {renling, yxy, devadas}@csail.mit.edu.

https://doi.org/10.1145/3177872

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. (2012) demonstrated that by observing accesses to an encrypted email repository, an adversary can infer as much as 80% of the search queries.

Oblivious RAM (ORAM) algorithms, first proposed by Goldreich (1987), Ostrovsky (1990), and Goldreich and Ostrovsky (1996), 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 a 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 (Ostrovsky and Shoup 1997; Williams et al. 2008; Williams and Sion 2008, 2012; Pinkas and Reinman 2010; Damgård et al. 2011; Boneh et al. 2011; Goodrich and Mitzenmacher 2011; Goodrich et al. 2012; Kushilevitz et al. 2012; Shi et al. 2011; Stefanov and Shi 2013b; Lorch et al. 2013; Williams et al. 2012; Stefanov et al. 2012).

In this article, 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. Our contributions for Path ORAM are stated as follows.

Theorem 1.1. *For a working set of N blocks where each block has size $B = \Omega(\log N)$ bits, with high probability, Path ORAM uses $O(B \log N + \log^2 N) \cdot \omega(1)$ bits of client storage and $O(B \log N + \log^3 N)$ bits of bandwidth.*

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 de-amortized oblivious sorting and oblivious cuckoo hash table construction like many existing ORAM algorithms (Goldreich and Ostrovsky 1996; Ostrovsky and Shoup 1997; Williams et al. 2008; Williams and Sion 2008, 2012; Pinkas and Reinman 2010; Damgård et al. 2011; Boneh et al. 2011; Goodrich and Mitzenmacher 2011; Goodrich et al. 2012; Kushilevitz et al. 2012).

Instead, each Path 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 polylogarithmic) client-side storage.

Asymptotic Efficiency. Table 1 compares the asymptotic efficiency of Path ORAM to prior works. For a reasonably large block size $B = \Omega(\log^2 N)$ bits, where N is the total number of blocks, and using nonuniform block sizes, (recursive) Path ORAM achieves an asymptotic *bandwidth overhead* of $O(\log N)$ blocks. (Recursion and nonuniform block size will be introduced in Section 4.) 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. Recursive Path ORAM consumes $O(\log N) \cdot \omega(1)$ blocks of client-side storage[2] to achieve a failure probability of $N^{-\omega(1)}$, negligible in N.

Path ORAM outperforms all prior schemes in bandwidth when the block size is at least $\Omega(\log^2 N)$ with nonuniform block sizes. For small block sizes $B = o(\log N \log\log N)$, Kushilevitz

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

[2] Throughout this article, 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 in base 2.

Table 1. Comparison to Other ORAM Schemes

ORAM Scheme	Data Block Size	Client Storage (# Blocks of Size B)	Read & Write Bandwidth (# Blocks of Size B)
Kushilevitz et al. (2012)	$B = \Omega(\log N)$	$O(1)$	$O(\log^2 N / \log\log N)$
Gentry et al. (2013) (recursive)	$B = \Omega(\log N)$	$O(\log^2 N) \cdot \omega(1)$	$O(\log^3 N / \log\log N) \cdot \omega(1)$
Chung et al. (2013) (recursive)	$B = \Omega(\log N)$	$O(\log^{2+\epsilon}(N))$	$O(\log^2 N \cdot \log\log N) \cdot \omega(1)$
Goodrich et al. (2012)	$B = \Omega(\log N)$	$O(N^\epsilon)$	$O(\log N)$
Path ORAM (recursive, nonuniform block size)	$B = \Omega(\log N)$	$O(\log N) \cdot \omega(1)$	$O(\log N + (\log N)^3/B)$

B is the block size in bits; N is the total number of blocks. For Path ORAM, we use nonuniform block sizes where data block size is B bits and position map block size is $\Theta(\log N)$ bits independent of B. The failure probability is set to $N^{-\omega(1)}$ in this table, i.e., negligible in N.

et al. have asymptotically smaller bandwidth overhead than Path ORAM. For very large block sizes $B = \Omega(N^\epsilon)$, Path ORAM matches Goodrich et al. (2012) but requires much less client storage. We note that the block-size requirement is not introduced by Path ORAM. All known ORAM algorithms effectively assume (often implicitly) a block size of at least $\Omega(\log N)$, since the block must be at least large enough to store the index of the block. Otherwise, if each block is a single bit, an additional $\log N$ factor should be multiplied to the cost for all known ORAM algorithms.

Practical and Theoretic Impact of Path ORAM. Since we first proposed Path ORAM (Stefanov and Shi 2012) 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. The Ascend secure processor (Fletcher et al. 2012) uses Path ORAM as a primitive, and later optimized Path ORAM for better hardware integration (Fletcher et al. 2015b, 2015c; Ren et al. 2017). Maas et al. (2013) 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. (2013), Chung and Pass (2013), and Chung et al. (2013). These three works also try to improve ORAM bounds based on the binary tree construction by Shi et al. (2011); however, as pointed out in Section 1.1, our bound is asymptotically better than those by Gentry et al. (2013), Chung and Pass (2013), and Chung et al. (2013). Gentry's Path ORAM variant construction has also been applied to secure multiparty computation (Gentry et al. 2013).

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 nonrecursive Path ORAM. For a nonrecursive Path ORAM and certain choices of parameters, we show that during a particular operation, the probability that the stash stores more than R blocks is at most $14 \cdot 0.6^{-R}$. Choosing $R = O(\log N) \cdot \omega(1)$ can make the failure probability negligible. Our empirical results in Section 7 indicate that the parameters in practice are even better than our theoretic bounds.

Our recursive Path ORAM construction has at most $\log N$ recursive ORAMs, so it needs at most $R \log N$ blocks of client storage. We shall do a more careful design and analysis to achieve $\Theta(R)$ blocks of client storage in the recursive construction.

1.1 Related Work

Oblivious RAM was first investigated by Ostrovsky (1990), Goldreich (1987), and Goldreich and Ostrovsky (1996) in the context of protecting software from piracy, and efficient simulation of programs on oblivious RAMs. Since then, there has been much subsequent work devoted to improving ORAM constructions (Ostrovsky and Shoup 1997; Williams et al. 2008; Williams and Sion 2008, 2012; Pinkas and Reinman 2010; Damgård et al. 2011; Boneh et al. 2011; Goodrich and Mitzenmacher 2011; Goodrich et al. 2012; Kushilevitz et al. 2012; Gentry et al. 2013; Chung and Pass 2013). Path ORAM is based on the binary-tree ORAM framework proposed by Shi et al. (2011).

Near Optimality of Path ORAM. Goldreich and Ostrovsky show that under $O(1)$ blocks of client storage, any ORAM algorithm that treats each block as an opaque ball (and hence works for generic block sizes) must have bandwidth cost at least $\Omega(\log N)$—even when the client can store arbitrary metadata (e.g., the entire position map) for free. 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 $O(\log N)$ block bandwidth cost (Goldreich and Ostrovsky 1996; Goodrich and Mitzenmacher 2011; Kushilevitz et al. 2012). Path ORAM is also the first ORAM construction to achieve $O(\log N)$ bandwidth overhead with small client-side storage (under any block size). This partially addresses this open question for reasonably large data block sizes and nonuniform block sizes. Note that the assumption of free metadata is stronger than large block and nonuniform block size. Hence, the Goldreich-Ostrovsky bound applies to large and nonuniform block sizes. We note that the subsequent work Circuit ORAM (Wang et al. 2015) improves our result by reducing the client-side storage to $O(1)$ blocks, showing that the Goldreich-Ostrovsky lower bound is tight in their specific model. It remains an open question whether the ORAM lower bound can be circumvented if the model is relaxed, e.g., if the block need not be treated as an opaque ball—Boyle and Naor (2016) have shown in subsequent work that this question is related to the circuit complexity for sorting.

Comparison with Gentry et al. and Chung et al. Gentry et al. (2013) improve on the binary tree ORAM scheme proposed by Shi et al. (2011). To achieve $2^{-\lambda}$ failure probability, their scheme achieves $O(\lambda(\log N)^2/(\log\lambda))$ block bandwidth cost, for block size $B = \Omega(\log N)$ bits. Adjusting the failure probability to $N^{-\omega(1)}$, i.e., negligible in N, their bandwidth cost is $O(\log^3 N/\log\log N)$ blocks.[3] Chung and Pass (2013) proved a similar (in fact, slightly worse) result as Gentry et al. (2013). Our result is asymptotically better than Gentry et al. (2013) or Chung and Pass (2013), as shown in Table 1.

In recent concurrent and independent work, Chung et al. (2013) proposed another statistically secure binary-tree ORAM algorithm 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[4] suggest an empirical bucket size of 4—which means that their practical bandwidth cost is a constant factor worse than Path ORAM, since they require operating on three paths in expectation for each data access, while Path ORAM requires reading and writing only one path.

Statistical Security. We note that Path ORAM is also statistically secure (not counting the encryption). Statistically secure or perfectly secure ORAMs have been studied in several prior works (Damgård et al. 2011; Ajtai 2010). All known binary-tree-based ORAM schemes and variants are also statistically secure (Shi et al. 2011; Chung and Pass 2013; Gentry et al. 2013; Ren et al. 2015) (assuming each bucket is a trivial ORAM).

[3]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.

[4]Personal communication with Kai-Min Chung.

Other Research Directions in ORAM. There has also been work to optimize ORAM for the number of communication rounds (Williams and Sion 2012; Fletcher et al. 2015a; Garg et al. 2016), response time (Dautrich et al. 2014), parallelism (Boyle et al. 2016; Chen et al. 2016), and various other metrics (Bindschaedler et al. 2015; Sahin et al. 2016). Some data structures can be made oblivious without using a full ORAM (Wang et al. 2014; Keller and Scholl 2014; Mitchell and Zimmerman 2014).

Applications of ORAM. ORAM has applications in many different theoretical and practical contexts. ORAM has often been quoted as a promising solution to privacy-preserving storage outsourcing (Williams et al. 2008, 2012; Stefanov and Shi 2013a, 2013b; Dautrich et al. 2014). ORAM has been adopted in the design of secure processors (Maas et al. 2013; Liu et al. 2015; Fletcher et al. 2012, 2015b, 2015c; Ren et al. 2017) to conceal the access pattern. ORAM has been frequently used as a building block in RAM-model cryptography such as garbled RAM (Lu and Ostrovsky 2013; Gentry et al. 2014; Garg et al. 2015a, 2015b; Canetti and Holmgren 2016; Canetti et al. 2015, 2016) and secure multiparty computation (Ostrovsky and Shoup 1997; Gordon et al. 2012; Boyle et al. 2015). It is well known that ORAM can eliminate access pattern leakage in searchable encryption (Islam et al. 2012; Curtmola et al. 2006), though some ORAM-like techniques (Naveed et al. 2014) were more often adopted for better efficiency. ORAM or ORAM-like techniques have been used to construct proofs of retrievability (Cash et al. 2013; Shi et al. 2013).

Relation to Private Information Retrieval. Private Information Retrieval (PIR) (Chor et al. 1998; Chor and Gilboa 1997; Ostrovsky and Skeith 2007) also hides access patterns from untrusted servers. Its main differences from ORAM are as follows. In PIR, a server holds public read-only data that should be accessible to anyone. In ORAM, a client outsources its private data to a server and can update the data. In PIR, the server can do computation on the data, while the traditional ORAM model assumes the server is a simple storage device that only supports read and write operations. A combination of the two has also been defined: a server stores private client data and is able to do computation over the data to reduce bandwidth overhead. This combination has been given multiple names, including private information storage (Ostrovsky and Shoup 1997), oblivious storage (Boneh et al. 2011), or simply ORAM with server computation (Devadas et al. 2016). Constructions with constant bandwidth overhead and polylogarithmic server computation have been shown in this model (Devadas et al. 2016).

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 to 256KB, while for secure processors smaller blocks (64B to 4KB) are preferable. Throughout the article, 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 Stefanov et al. (2012). 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 2.1 (Security Definition). Let

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

denote a data request sequence of length M, where each op_i denotes a $\mathrm{read}(\mathrm{a}_i)$ or a $\mathrm{write}(\mathrm{a}_i, \mathrm{data})$ 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 $1 - negl(|\vec{y}|)$; i.e., the ORAM may fail with probability $negl(|\vec{y}|)$.

Adaptive Versus Nonadaptive Access Sequence. In the above security definition, an adversary chooses the request sequence upfront and the notion seems to cover only a nonadaptive request sequence. However, as we shall see, in our protocols, fresh randomness is generated for each access. Hence, our proofs actually imply that there is a probabilistic polynomial-time simulator that sees only the length but not the request sequence, and its output is statistically close to the sequence of accesses made by our ORAM algorithm upon any *adaptively* chosen request sequence.

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: NONRECURSIVE VERSION

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.

Path. Let $x \in \{0, 1, \ldots, 2^L - 1\}$ denote the xth 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 the 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.

Bucket. Each node in the tree is called a bucket. Each bucket can contain up to Z real blocks. Each real block is stored alongside some metadata: the address a of the block and the leaf x it is mapped to. This forms a 3-tuple $(\mathrm{a}, x, \mathrm{data})$. If a bucket has fewer than Z real blocks, it is padded with dummy blocks (with dummy metadata) 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). Since there are $2^L - 1$ buckets in the tree, the total server storage used is about $Z \cdot 2^L$ blocks.

3.3 Client Storage and Bandwidth

The storage on the client consists of two 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 xth 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. Note that the mapping is also stored as metadata for each block in the binary tree on the server side.

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 $2Z \log N$ block 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 nonrecursive Path ORAM is at most $O(\log N)\omega(1)$ blocks to obtain negligible failure probability. Later 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 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 data ← Access

Access(op, a, data*):

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

Fig. 1. **Protocol for data access.** Read or write a data block identified by a. If op = read, the input parameter data* = 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.

Table 2. Notations

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

(read, a, None), and to write data* to block a, the client performs Access(write, a, data*). The Access protocol can be summarized in four 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. If the client performs Access on block a for the first time, it will not find block a in the tree or stash, and should assume that the block has a default value of zero.
(3) **Update block** (Lines 6 to 9): If the access is a write, update the data of 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′ mapped to leaf x' can be placed in the bucket at

level ℓ only if the path $\mathcal{P}(x')$ intersects the path accessed $\mathcal{P}(x)$ at level ℓ, in other words, if $\mathcal{P}(x, \ell) = \mathcal{P}(x', \ell)$.

Subroutines. For ReadBucket(bucket), the client reads all Z blocks and their metadata (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 with their metadata 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 storage device, so it only performs operations 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

$$\mathbf{p} = (x_M, x_{M-1}, \ldots, x_1),$$

where $x_j = \text{position}_j[\mathsf{a}_j](1 \le j \le M)$ is the position of address a_j indicated by the position map for the jth load/store operation. The order of accesses from M to 1 follows the notation from Definition 2.1. The server also sees a sequence of encrypted paths $\mathcal{P}(x_j)$, which is computationally indistinguishable from a random sequence of bit strings due to randomized encryption.

Notice that once x_i is revealed to the server, it is remapped to a completely new random label; hence, x_i is statistically independent of any x_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, x_i is statistically independent of x_j for $j < i$ with $\mathsf{a}_j \neq \mathsf{a}_i$. This shows that x_i is statistically independent of x_j for $j < i$. Therefore, by Bayes rule, $\Pr(\mathbf{p}) = \prod_{j=1}^{M} \Pr(x_j) = (\frac{1}{2^L})^M$. This proves that $A(\vec{y})$ is computationally indistinguishable from a random sequence of bit strings.

Now the security follows from Theorem 5.1 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 the scheme described in the previous section, the client must store a relatively large position map. We leverage the recursion idea as described in the ORAM constructions of Stefanov et al. (2012) and Shi et al. (2011) 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 $\text{ORam}_0, \text{ORam}_1, \text{ORam}_2, \ldots, \text{ORam}_X$, 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. We call ORam_0 the data ORAM, and all the others position map ORAMs. An important parameter is the block size for position map ORAMs, or equivalent, the number of position map entries each block can contain. If we use a block size of $\chi \log N$ for position map ORAMs, each block can contain χ position map entries for the previous ORAM. Clearly, we require $\chi \ge 2$, so that ORam_{i+1} is smaller than

ORam_i (by a factor of χ). Then, for a block a_0 in ORam_0, its position is stored in block $\text{a}_1 = \frac{\text{a}_0}{\chi}$ of ORam_1, whose position is in turn stored in block $\text{a}_1 = \frac{\text{a}_0}{\chi^2}$ in ORam_2 (all divisions take a floor on the results), and so on. After $\frac{\log N}{\log \chi}$ levels of recursion, the final position map has constant size.

Now to access block a_0 in ORam_0, the client looks up and updates its position in block a_1 in ORam_1, which triggers a recursive call to look up and update a_1's position in block a_2 in ORam_2, and so on until finally a position of ORam_χ is looked up and updated in the client storage. Essentially, this replaces lines 1 and 2 in Figure 1 for ORam_i with a recursive call to Access in ORam_{i+1} with input (write, $\frac{\text{a}_i}{\chi}, x^*$). Here, we need a write access in ORam_{i+1} to return data and also support partial write to a block (since we are updating one position map entry in a block). The rest of the pseudocode in Figure 1 does not use position map and remains unchanged.

4.2 Parameterization and Other Metrics

In this section, we consider different ways to parameterize the scheme and additional metrics. In general, we consider two settings, nonuniform and uniform block sizes. Nonuniform block size allows the data blocks and metadata blocks (i.e., position map levels) to have different sizes, whereas uniform block size requires that they have the same size. Besides the bandwidth metric, we also consider (1) the number of accesses (in terms of blocks fetched)—this translates to the runtime of the oblivious RAM program if we count each memory access as one time step; and (2) number of roundtrips—assuming that each roundtrip can read and write multiple blocks.

- —*Nonuniform block size.* We suggest parameterizing the scheme such that each position map block has $\chi \log N$ bits. In this case, the total bandwidth consumed for accessing a block is $O(B \log N + \log^3 N)$ bits, and if $B = \Omega(\log^2 N)$, the bandwidth blowup factor is $O(\log N)$. In this case, the total number of accesses is $O(\log^2 N)$, and the number of roundtrips is $O(\log N)$.
- —*Uniform block size.* In this case, each position map block has the same size as the data block and can store $\chi := \frac{B}{\log N}$ position map entries. Therefore, the number of position map ORAMs is $O(\frac{\log N}{\log \chi})$. We conclude that the number of roundtrips is $O(\frac{\log N}{\log \chi})$, the total number of accesses is $O(\frac{\log^2 N}{\log \chi})$, and the total bandwidth consumed for accessing a block is $O(\frac{B \log^2 N}{\log \chi})$. As a special case, if the block size is reasonably large, say, $B = N^\epsilon$ for some constant $0 < \epsilon < 1$, then the number of position map ORAMs is $O(1)$, and the bandwidth required to read one block is $O(B \log N)$.

5 THEORETIC BOUNDS ON STASH USAGE

In this section, we will analyze the stash usage for a nonrecursive 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 stash exceeds R, and show that this probability decreases exponentially in R.

By ORAM_L^Z we denote a nonrecursive 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 $\text{ORAM}_L^Z[\mathbf{s}]$ 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 $\text{st}(\text{ORAM}_L^Z[\mathbf{s}])$ 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 $\text{st}(\text{ORAM}_L^Z[\mathbf{s}]) > R$.

Theorem 5.1 (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*

$$\Pr\left(\text{st}\left(\text{ORAM}_L^5[\mathbf{s}]\right) > R \mid a(\mathbf{s}) = \mathbf{a}\right) \leq 14 \cdot 0.6^R,$$

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 $\leq s \cdot 14 \cdot 0.6^R$. 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 postprocesses 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 postprocessing leads to a stash usage $> R$. We show that the stash usage after postprocessing 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 $\Pr[\text{st}(\text{ORAM}_L^Z[\mathbf{s}]) > R \mid a[\mathbf{s}] = \mathbf{a}\,)$ 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 *postprocessing* 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 $\text{st}^Z(\text{ORAM}_L^\infty[\mathbf{s}])$ to denote the stash usage

after the greedy algorithm is applied. The greedy algorithm is defined in terms of the following operations.

Defining Operations to Analyze Bucket Usage. In order to compare the ∞-ORAM and the real Path ORAM, we define the following operations that can be applied to both ORAMs. We remark that the purpose of these operations is for analyzing bucket usage and they are by no means related to hiding data access patterns.

(1) Given a leaf label x, evict(x) performs eviction along the path from the root to the leaf x as in the ∞-ORAM.
(2) Given a leaf label x, pull(x) performs a procedure starting from the leaf bucket x along the path toward the root bucket to ensure the usage of each bucket does not exceed its capacity Z. Specifically, when we process a bucket whose capacity is smaller than the number of blocks it contains, we assume that there is some **global ordering** on the block ids to consistently decide which blocks have to be moved to the parent for further processing.
(3) Given a block id a, update(a) changes the position map of block a with a new label and moves the block in the root.

Therefore, we can express accessing a block a with x = position[a] in terms of these operations in each of the cases as follows:

(1) For the **real Path ORAM**: update(a), evict(x), pull(x).
(2) For the **∞-ORAM**: update(a), evict(x).

We say that two sequences of operations are *equivalent* if, starting from any configuration of the ORAM, each bucket contains exactly the same blocks (not just the same number) after the execution of the two sequences.

Lemma 5.2. *Given any leaf labels x and y, the following two sequences are equivalent:*

(a) pull(x), pull(y)
(b) pull(y), pull(x)

Proof. Consider any initial configuration, and let C be the lowest common ancestor of x and y. Since there is a global ordering on the block ids to decide which blocks to keep in a bucket during processing, both sequences are equivalent to the following procedure:

(1) Perform postprocessing from x toward C up to the point when blocks are added to C.
(2) Perform postprocessing from y toward C up to the point when blocks are added to C.
(3) Perform postprocessing from C toward the root. □

Defining Postprocessing Algorithm G_Z. In view of Lemma 5.2, the greedy postprocessing G_Z is defined as concatenating operations pull(x) for all leaves x in any order.

Lemma 5.3. *Suppose the block* a *has label x* = position[a], *and y is any label. Then, the following two sequences are equivalent:*

(a) update(a), evict(x), pull(x), pull(y)
(b) pull(y), update(a), evict(x), pull(x)

Proof. Since Lemma 5.2 states that the pull operations are commutative, sequence (a) is equivalent to
(a') update(a), evict(x), pull(y), pull(x).

We next argue why (a') is equivalent to (b). The strategy is that we start from some fixed configuration σ_0 of the ORAM and run the process described in sequence (a') and pause it at some instant. Suppose C is the lowest common ancestor of the leaves x and y. Specifically, we run the operations in (a'), and during the execution of pull(y), we pause the process at the moment when the set Q of blocks originating from the path from leaf y to C are added to the bucket C. We denote the configuration of the ORAM at this instant by σ_1. Observe that after pull(y) in (a'), we will immediately have pull(x). Hence, it is actually not necessary to complete the execution of pull(y) and we break this operation. We will resume the execution of (a') later, and note that there is only pull(x) to be performed.

We next start from configuration σ_0 and start executing sequence (b). While executing operation pull(y), we pause the process at the moment when the set Q of blocks is added to C, which we recall is the lowest common ancestor of x and y. Observe that the next operation is evict(x) in (b). Hence, there is no need to continue the execution of pull(y) for processing from C to the root, because any work done will be immediately nullified by the next evict(x). We next execute evict(x) and observe that for an evict operation, blocks are oblivious to one another because the position of a block is independent of whether other blocks are present. Since all the blocks in Q will stay at C, it follows that after executing evict(x) in (b), the configuration of the ORAM is also σ_1, which is exactly the configuration reached at the instant we have paused the execution of (a').

We next can resume executing both sequences. Since at this instant both executions are at configuration σ_1, and the remaining operation in both executions is pull(x), it follows that both executions must reach the same configuration afterward. □

Lemma 5.4. *The stash usage in a postprocessed ∞-ORAM is exactly the same as the stash usage in Path ORAM:*

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

Proof. We observe that in the real Path ORAM, the operations evict and pull are interleaved, while postprocessing an ∞-ORAM can be interpreted as having the evict operations first, followed by all possible pull operations. Specifically, suppose the operation sequence corresponding to n accesses in the real Path ORAM is:

(a): update(a_n), evict(x_n), pull(x_n), ..., update(a_1), evict(x_1), pull(x_1).

By applying Lemma 5.3 repeatedly, we have the following equivalent sequence:

(b): update(a_n), evict(x_n), ..., update(a_1), evict(x_1), pull(x_1), ..., pull(x_n).

We remark that the pull operations are in reversed order, but actually it does not really matter because of Lemma 5.2.

Observe that the ∞-ORAM before the postprocessing can be obtained by the sequence

(c): update(a_n), evict(x_n), ..., update(a_1), evict(x_1).

After appending (c) with a concatenation of pull(x) operations for all leaves x, we will have a postprocessed ∞-ORAM.

The final step is that in sequence (b), a pull(x) operation might be missing for some leaf x, because there is no corresponding evict(x) operation. Observe that at the end of sequence (b), no bucket will have its capacity violated. Hence, adding the operations pull(x) for all missing leaves x to sequence (b) at the end has no effect. This completes the proof. □

5.2 Usage/Capacity Bounds

To investigate when a not-processed ORAM_L^∞ can lead to a stash usage of $>R$ after postprocessing, 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 $\mathrm{u}^T(\mathrm{ORAM}_L^\infty[\mathbf{s}])$ 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 5.5. *The stash usage* $\mathrm{st}^Z(\mathrm{ORAM}_L^\infty[\mathbf{s}])$ *in postprocessed* ∞*-ORAM is* $>R$ *if and only if there exists a subtree* T *in* ORAM_L^∞ *such that* $\mathrm{u}^T(\mathrm{ORAM}_L^\infty[\mathbf{s}]) > n(T)\cdot Z + R$.

Proof.

If part: Suppose T is a subtree such that $\mathrm{u}^T(\mathrm{ORAM}_L^\infty[\mathbf{s}]) > n(T)\cdot Z + R$. 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 $\mathrm{st}^Z(\mathrm{ORAM}_L^\infty[\mathbf{s}]) > R$. Define T to be the maximal subtree that contains only buckets with exactly Z blocks after postprocessing 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 fewer than Z blocks after postprocessing, 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 $\mathrm{u}^T(\mathrm{ORAM}_L^\infty[\mathbf{s}]) > n(T)\cdot Z + R$ □

Lemma 5.6 (Worst-case Address Pattern). *Fix some subtree* T *and positive integer* R*. Out of all address sequences* $\mathbf{a}$ *such that the number of distinct blocks accessed is* N*, the probability* $\Pr[\mathrm{u}^T(\mathrm{ORAM}_L^\infty[\mathbf{s}]) > R | a(\mathbf{s}) = \mathbf{a}]$ *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 jth 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 bucket outside T; also notice that buckets in ∞-ORAM are infinitely sized, so removing the jth 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 $\Pr[\mathrm{u}^T(\mathrm{ORAM}_L^\infty[\mathbf{s}]) > R | a(\mathbf{s}) = \mathbf{a}]$ is maximized when $\mathbf{a} = (a_i)_{i=1}^s$ is a sequence of block addresses without duplicates. □

Remark 5.7. Observe that Lemma 5.6 only implies that the worst-case address pattern consists of accesses to distinct blocks. However, it might be possible that the worst case is achieved when the number of distinct blocks is less than N. While we do not directly claim that accessing N distinct blocks must be the worst case, we remark that our probability bound will still hold even if the number of distinct blocks accessed is smaller than N. Specifically, the number of distinct blocks only plays a part in the *Balls-and-Bins Game* that we define in Section 5.5.

Bounding Usage for Each Subtree. In view of Lemma 5.6, 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 5.4 and 5.5, we obtain

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

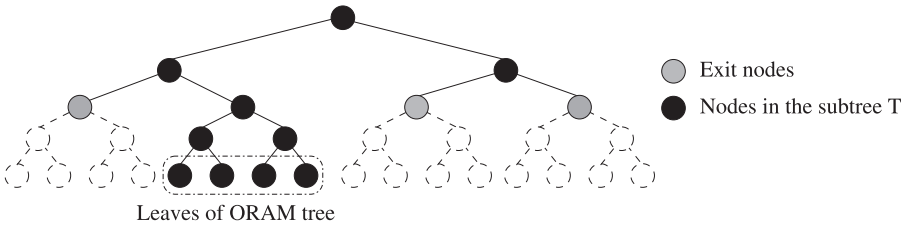


Fig. 2. A subtree containing some leaves of the original ORAM binary tree, augmented with the exit nodes.

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 $\leq 4^n$,

$$\Pr\left[\mathrm{st}\left(\mathrm{ORAM}_L^Z[\mathbf{s}]\right) > R\right] \leq \sum_{n\geq 1} 4^n \max_{T:n(T)=n} \Pr\left[\mathrm{u}^T\left(\mathrm{ORAM}_L^\infty[\mathbf{s}]\right) > nZ + R\right]. \tag{1}$$

We next give a uniform upper bound for $\Pr[\mathrm{u}^T(\mathrm{ORAM}_L^\infty[\mathbf{s}]) > nZ + R]$ in terms of n, Z, and R.

5.3 Analyzing Usage of Subtree via Eviction Game

In view of Lemma 5.6, 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 reverse 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* $P_{\mathrm{evict}}(i)$.

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

Recall that given subtree T, we want to probabilistically bound the quantity $u^T(\mathrm{ORAM}_L^\infty[\mathbf{s}])$, 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 5.8 (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 $u := \mathrm{exit}(P, T)$. If the whole path P is contained in T, then the exit node $\mathrm{exit}(P, T)$ is *null*.

To bound the number of blocks residing in T at the end of an access sequence of N, we use the following lemma.

Lemma 5.9. *Suppose at step i, a block a_i is requested and is assigned to some path $P_{assign}(i)$ at the end of the request. Then, the block a_i will be evicted from tree T* if and only if *there exists a later step $j \leq i$ (recalling that smaller indices mean later) such that both exit nodes $\mathrm{exit}(P_{evict}(j), T)$ and $\mathrm{exit}(P_{assign}(i), T)$ are equal and not null.*

Proof. Observe that after step i, block a_i can only be evicted by going through the exit node $\mathrm{exit}(P_{\mathrm{assign}}(i), T)$, if it is not null.

Hence, block a_i is evicted *if and only if* in some later step j, the path $P_{\mathrm{evict}}(j)$ intersects $\mathrm{exit}(P_{\mathrm{assign}}(i), T)$, which is equivalent to the conclusion of the lemma. □

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 the following table.

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

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 *if and only if* the path $P_{\text{assign}}(i)$ intersects a node in F (and block b_i survives), and Y_i to be the indicator variable that equals 1 *if and only if* 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 5.10. *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_is 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 is. Hence, it follows from

Dubhashi and Ranjan (1998, Proposition 7(1)) that, conditioning on $\mathbf{q}$, X and Y are negatively associated.

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

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 $Y \le \widehat{Y}$.

(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. As mentioned in Remark 5.7, if the number of distinct blocks is smaller than N, then in this game, the number of balls will be smaller. Hence, the corresponding random variable will be stochastically dominated by the case with N balls.

(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 toward Y, but might be counted toward $\widehat{Y}$ 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 nonnegative t, $E[e^{tY}] \le E[e^{t\widehat{Y}}]$.

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 5.11. *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 $R > 0$, $\Pr[\mathrm{u}^T(\mathrm{ORAM}_L^\infty[\mathbf{s}]) > n \cdot Z + R] \le \frac{1}{4^n} \cdot (0.9332)^n \cdot e^{-0.5105R}$.*

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, $E[e^{tX_i}] = (1 - \frac{l}{N}) + \frac{l}{N}e^t \le \exp(\frac{l}{N}(e^t - 1))$, and hence by independence, $E[e^{t\widehat{X}}] \le \exp(l(e^t - 1))$.

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 in the first round of the infinite eviction game, the first eviction path opens one of the nodes in $\widehat{E}$, and hence, the number of open nodes for the first block is 1.) 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; specifically, $\Pr[Q_j = i | q_j] = (1 - q_j)^i q_j$, for $i \geq 0$.

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 $\mathbf{w} := \{w_j\}_{j \leq k}$, which 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 $\mathbf{w} = \{w_j\}_{j \leq k}$ 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:

$$E[e^{tM}|\mathbf{w}] = \sum_{i \geq 0} q(1-q)^i E[e^{tM}|Q = i, \mathbf{w}] \tag{2}$$

$$\leq \sum_{i \geq 0} q(1-q)^i \exp(qi(e^t - 1)) \tag{3}$$

$$\leq \frac{q \exp(q(e^t - 1))}{1 - (1-q)\exp(q(e^t - 1))} \tag{4}$$

$$= \frac{q}{\exp(-q(e^t - 1)) - (1-q)} \tag{5}$$

$$\leq \frac{q}{1 - q(e^t - 1) - 1 + q} \tag{6}$$

$$= \frac{1}{2 - e^t}. \tag{7}$$

From Equations (2) to (3), we consider the moment-generating function of a sum of i independent Bernoulli random variables, each having expectation q: $E[e^{tM}|Q = i, \mathbf{w}] = ((1-q) + qe^t)^i \leq \exp(qi(e^t - 1))$. In Equation (4), for the series to converge, we observe that $(1-q)\exp(q(e^t - 1)) \leq \exp(q(e^t - 2))$, which is smaller than 1 when $0 < t < \ln 2$. In Equation (6), we use $1 - u \leq \exp(-u)$ 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 $E[e^{t\sum_{j=1}^{k} M_j}] \leq E[e^{t\sum_{j=1}^{k-1} M_j}] \cdot (\frac{1}{2-e^t})$.

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 $1 < \kappa \leq k$, we have $E[e^{t\sum_{j=1}^{\kappa} M_j}] \leq E[e^{t\sum_{j=1}^{\kappa-1} M_j}] \cdot (\frac{1}{2-e^t})$.

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 of 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 $t = \ln \frac{5}{3}$, one can check that $\max\{\frac{e^{-tZ}}{2-e^t}, e^{e^t-1-tZ}\} \le \frac{1}{4} \cdot 0.93312$, and so the probability is at most $\frac{1}{4^n} \cdot (0.93312)^n \cdot 0.6^R$. □

Proof of Theorem 5.1. By applying Lemma 5.11 to the inequality in Equation (1), we have the following: $\Pr[\mathrm{st}(\mathrm{ORAM}_L^Z[\mathbf{s}]) > R | a(\mathbf{s}) = \mathbf{a}] \le \sum_{n\ge 1} 4^n \cdot \frac{1}{4^n} \cdot (0.93312)^n \cdot 0.6^R \le 14 \cdot 0.6^R$, 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 5.1, for each $k \in [K]$, for each $R > 0$, $\Pr[S_k > R] \le 14 \cdot 0.6^R$. Observing that a geometric distribution G with parameter p satisfies $\Pr[G > R] \le (1-p)^R$, we have the following.

Proposition 5.12. *For each $k \in [K]$, the random variable S_k is stochastically dominated by $3 + G$, where G is the geometric distribution with parameter $p = 1 - 0.6 = 0.4$.*

From Proposition 5.12, 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_ks are independent geometric distribution with parameter $p = 0.4$. 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 5.13 (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 $\Pr[\sum_{k\in[K]} G_k > E[\sum_{k\in[K]} G_k] + R] \le \exp(-\frac{pR}{2} + \frac{K}{2})$.*

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

$$E[e^{tG_k}] = \sum_{i\ge 1} p(1-p)^i \cdot e^{it} \tag{8}$$

$$= \frac{pe^t}{1-(1-p)e^t} = \frac{p}{p+e^{-t}-1} \tag{9}$$

$$\le \frac{p}{p-t} = \frac{1}{1-\frac{t}{p}} \tag{10}$$

$$\leq 1 + \frac{t}{p} + \frac{2t^2}{p^2} \tag{11}$$

$$\leq \exp\left(\frac{t}{p} + \frac{2t^2}{p^2}\right), \tag{12}$$

where in Equation (9), the geometric series converges, because $t \leq \frac{p}{2} < \ln \frac{1}{1-p}$, for $0 < p < 1$. In Equation (10), we use the inequality $1 - e^{-t} \leq t$; in Equation (11), 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 5.13, 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. (2013) propose a novel method for performing an entire binary search using a single ORAM lookup. 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. (2012) introduce the concept of *stateless* ORAM, where the client state is small enough so that any client accessing the ORAM can download it before each data access and upload it afterward. 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

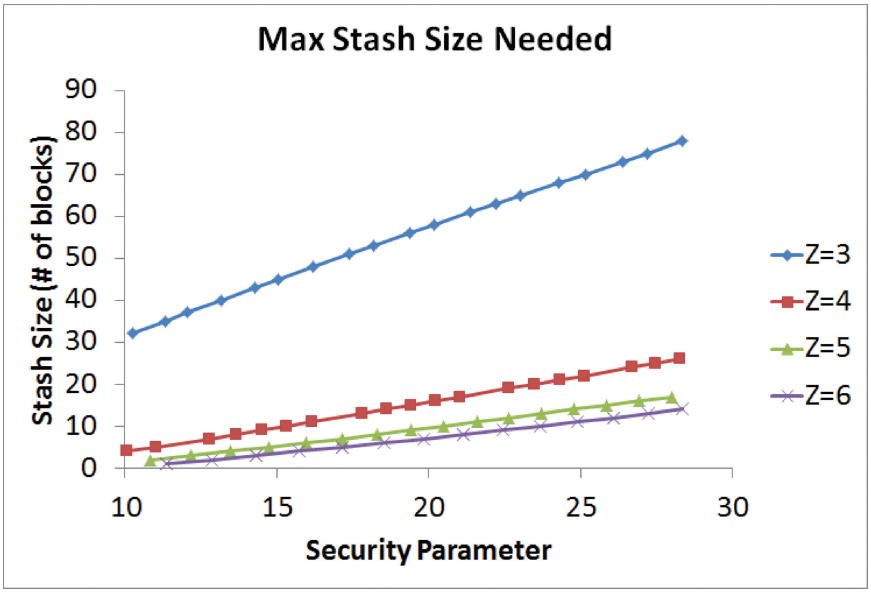


Fig. 3. Empirical estimation of the required stash size to achieve failure probability less than $2^{-\lambda}$, where λ is the security parameter. Measured for $N = 2^{16}$, but as Figure 4 shows, the stash size does not depend on N (at least for $Z = 4$). The measurements represent a worst-case (in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

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

The Ascend secure processor (Fletcher et al. 2012; Ren et al. 2013b; Fletcher et al. 2015b, 2015c; Ren et al. 2017) built Path ORAM in both FPGA and custom silicon. They improve the bandwidth cost and stash size of the recursive construction through various hardware and architectural optimizations. Maas et al. (2013) built a hardware implementation of a Path ORAM-based secure processor using FPGAs and the Convey platform. Both designs rely on on-chip caches while making Path ORAM requests only when last-level cache misses occur. The two projects report about 1.2× to 5× performance overhead for many benchmarks such as SPEC traces and SQLite queries.

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 path, 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. Ren et al. (2013a) and Fletcher et al. (2015b) further optimize the integrity verification overhead for the recursive Path ORAM construction.

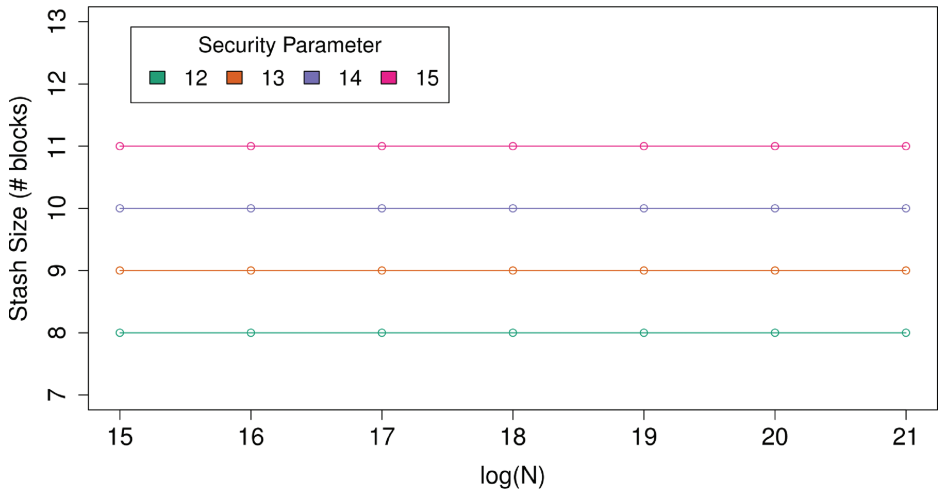


Fig. 4. The stash size to achieve failure probability less than $2^{-\lambda}$ does not depend on N ($Z = 4$). Measured for a worst-case (in terms of stash size) access pattern. The stash size does not include the temporarily fetched path during Access.

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

Fig. 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.

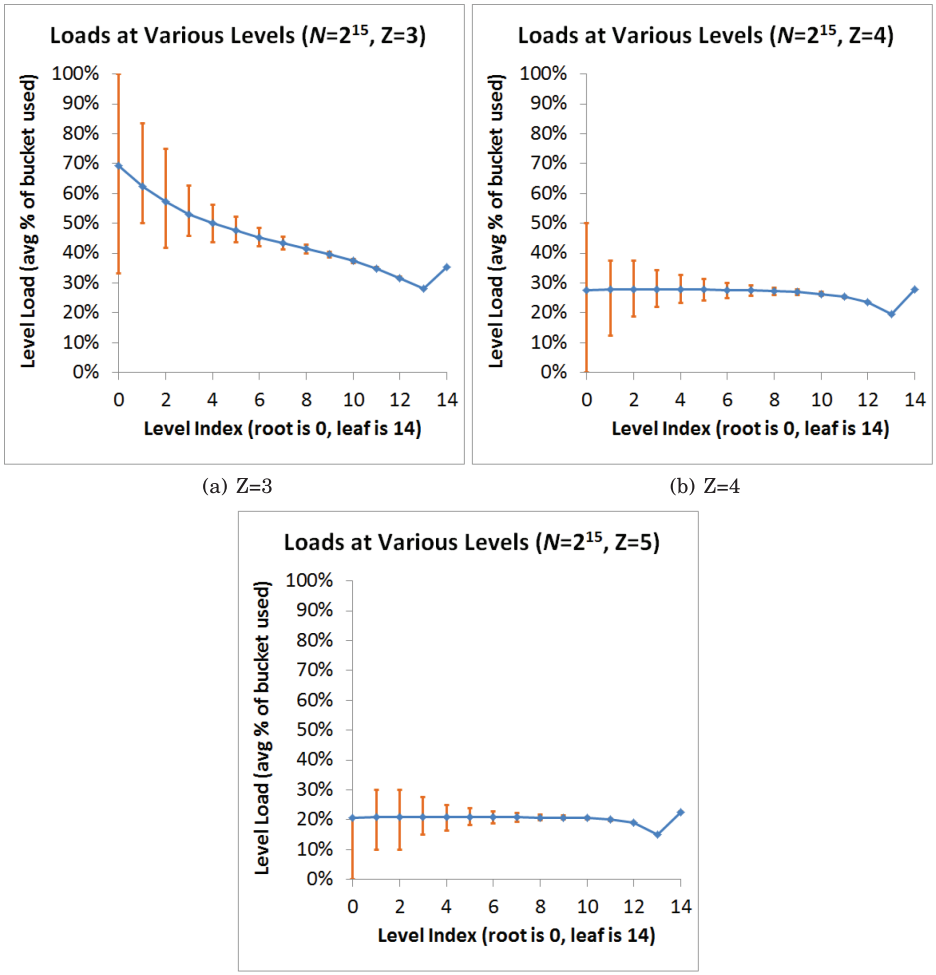
7 EVALUATION

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 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 that 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. That is, 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



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

(c) Z=5

Fig. 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.

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 Harchol-Balter (2013)).

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, which is in accordance with the main theorem in Section 5 that the failure probability decreases exponentially with the stash size. Figure 4 shows that the required stash size for a low failure probability ($2^{-\lambda}$) does not depend on N. This shows that 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. (2013).

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 one 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).

ACKNOWLEDGMENTS

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

REFERENCES

Miklós Ajtai. 2010. Oblivious RAMs without cryptogrpahic assumptions. In *Proceedings of the 42nd ACM Symposium on Theory of Computing (STOC'10)*. ACM, 181–190.

Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. 2015. Practicing oblivious access on cloud storage: The gap, the fallacy, and the new way forward. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 837–849.

Dan Boneh, David Mazieres, and Raluca Ada Popa. 2011. Remote Oblivious Storage: Making Oblivious RAM practical. Manuscript. Retrieved from http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf.

Elette Boyle, Kai-Min Chung, and Rafael Pass. 2015. Large-scale secure computation: Multi-party computation for (parallel) RAM programs. In *Advances in Cryptology - Proceedings of the 35th Annual Cryptology Conference (CRYPTO'15), Part II*. Springer, 742–762.

Elette Boyle, Kai-Min Chung, and Rafael Pass. 2016. Oblivious parallel RAM and applications. In *Theory of Cryptography Conference*. Springer, 175–204.

Elette Boyle and Moni Naor. 2016. Is there an oblivious RAM lower bound? In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*. ACM, 357–368.

Ran Canetti, Yilei Chen, Justin Holmgren, and Mariana Raykova. 2016. Adaptive succinct garbled RAM or: How to delegate your database. In *Theory of Cryptography Conference*. Springer, 61–90.

Ran Canetti and Justin Holmgren. 2016. Fully succinct garbled RAM. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science (ITCS'16)*. ACM.

Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan. 2015. Succinct garbling and indistinguishability obfuscation for RAM programs. In *Proceedings of the 47th Annual ACM Symposium on Theory of Computing (STOC'15)*. ACM, 429–437.

David Cash, Alptekin Küpçü, and Daniel Wichs. 2013. Dynamic proofs of retrievability via oblivious RAM. In *Advances in Cryptology (EUROCRYPT'13)*. Springer, 279–295.

Binyi Chen, Huijia Lin, and Stefano Tessaro. 2016. Oblivious parallel ram: Improved efficiency and generic constructions. In *Theory of Cryptography Conference*. Springer, 205–234.

Benny Chor and Niv Gilboa. 1997. Computationally private information retrieval (extended abstract). In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC'97)*. ACM, 304–313.

Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. 1998. Private information retrieval. *Journal of the ACM (JACM)* 45, 6 (1998), 965–981.

Kai-Min Chung, Zhenming Liu, and Rafael Pass. 2013. Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ Overhead. Retrieved from http://arxiv.org/abs/1307.3699.

Kai-Min Chung and Rafael Pass. 2013. A Simple ORAM. Retrieved from https://eprint.iacr.org/2013/243.pdf.

Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable symmetric encryption: Improved definitions and efficient constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*. ACM, 79–88.

Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. 2011. Perfectly secure oblivious RAM without random oracles. In *Proceedings of the 8th Conference on Theory of Cryptography (TCC'11)*. Springer-Verlag, Berlin, 144–163.

Jonathan Dautrich, Emil Stefanov, and Elaine Shi. 2014. Burst ORAM: Minimizing ORAM response times for bursty access patterns. In *23rd USENIX Security Symposium*. USENIX Association, 749–764.

Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. 2016. Onion ORAM: A constant bandwidth blowup oblivious ORAM. In *Theory of Cryptography*. Springer, 145–174.

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

Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. 2012. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the 7th ACM Workshop on Scalable Trusted Computing*. ACM, 3–8.

Christopher W. Fletcher, Muhammad Naveed, Ling Ren, Elaine Shi, and Emil Stefanov. 2015a. Bucket ORAM: Single online roundtrip, constant bandwidth oblivious RAM. *IACR Cryptology ePrint Archive* (2015), 1065.

Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas Devadas. 2015b. Freecursive oram: [Nearly] free recursion and integrity verification for position-based oblivious RAM. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 103–116.

Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, Emil Stefanov, Dimitrios Serpanos, and Srinivas Devadas. 2015c. A low-latency, low-area hardware oblivious RAM controller. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'15)*. IEEE, 215–222.

Sanjam Garg, Steve Lu, and Rafail Ostrovsky. 2015a. Black-box garbled RAM. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science (FOCS'15)*. IEEE, 210–229.

Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. 2015b. Garbled RAM from one-way functions. In *Proceedings of the 47th Annual ACM on Symposium on Theory of Computing*. ACM, 449–458.

Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. 2016. TWORAM: Efficient oblivious RAM in two rounds with applications to searchable encryption. In *Annual Cryptology Conference*. Springer, 563–592.

Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit Julta, Mariana Raykova, and Daniel Wichs. 2013. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies*. Springer, 1–18.

Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. 2014. Garbled RAM revisited. In *Advances in Cryptology - EUROCRYPT - International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 405–422.

Oded Goldreich. 1987. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*. ACM, 182–194.

Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)* 43 (1996), 431–473.

Michael T. Goodrich and Michael Mitzenmacher. 2011. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *Automata, Languages and Programming*. Springer, 576–587.

Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. 2012. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*. SIAM, 157–167.

S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. 2012. Secure two-party computation in sublinear (amortized) time. In *ACM Conference on Computer and Communications Security (CCS'12)*. ACM.

Mor Harchol-Balter. 2013. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press.

Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, Vol. 20, 12.

Marcel Keller and Peter Scholl. 2014. Efficient, oblivious data structures for MPC. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 506–525.

Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. 2012. On the (in) security of hash-based oblivious RAM and a new balancing scheme. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 143–156.

Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. 2015. GhostRider: A hardware-software system for memory trace oblivious computation. *SIGPLAN Notices* 50, 4 (March 2015), 87–101.

Jacob R. Lorch, Bryan Parno, James W. Mickens, Mariana Raykova, and Joshua Schiffman. 2013. Shroud: Ensuring private access to large-scale data in the data center. *FAST* (2013), 199–213.

Steve Lu and Rafail Ostrovsky. 2013. How to garble RAM programs. In *EUROCRYPT*. Springer.

Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. 2013. PHANTOM: Practical oblivious computation in a secure processor. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS'13)*. ACM.

John C. Mitchell and Joe Zimmerman. 2014. Data-oblivious data structures. In *Theoretical Aspects of Computer Science (STACS'14)*.

Muhammad Naveed, Manoj Prabhakaran, and Carl A. Gunter. 2014. Dynamic searchable encryption via blind storage. In *IEEE Symposium on Security and Privacy (SP'14)*. IEEE, 639–654.

Rafail Ostrovsky. 1990. Efficient computation on oblivious RAMs. In *STOC*. ACM, 514–523.

Rafail Ostrovsky and Victor Shoup. 1997. Private information storage. In *STOC*. ACM, 294–303.

Rafail Ostrovsky and William E. Skeith, III.2007. A survey of single-database private information retrieval: Techniques and applications. In *Proceedings of the 10th International Conference on Practice and Theory in Public-Key Cryptography (PKC'07)*. Springer-Verlag, 393–411.

Benny Pinkas and Tzachy Reinman. 2010. Oblivious RAM revisited. In *Advances in Cryptology (CRYPTO'10)*. Springer, 502–519.

Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. 2015. Constants count: Practical improvements to oblivious RAM. In *24th USENIX Security Symposium (USENIX Security'15)*. USENIX Association, 415–430.

Ling Ren, Christopher W. Fletcher, Albert Kwon, Marten van Dijk, and Srinivas Devadas. 2017. Design and implementation of the ascend secure processor. *IEEE Transactions on Dependable and Secure Computing* PP, 99 (2017).

Ling Ren, Christopher W. Fletcher, Xiangyao Yu, Marius van Dijk, and Srinivas Devadas. 2013a. Integrity verification for path oblivious-RAM. In *HPEC*. IEEE, 1–6.

Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten Van Dijk, and Srinivas Devadas. 2013b. Design space exploration and optimization of path oblivious RAM in secure processors. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 571–582.

Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Rachel Lin, and Stefano Tessaro. 2016. TaoStore: Overcoming asynchronicity in oblivious data storage. In *IEEE Symposium on Security and Privacy (SP'16)*.

Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*. Springer, 197–214.

Elaine Shi, Emil Stefanov, and Charalampos Papamanthou. 2013. Practical dynamic proofs of retrievability. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 325–336.

Emil Stefanov and Elaine Shi. 2012. Path O-RAM: An extremely simple oblivious RAM protocol. *CoRR* abs/1202.5150 (2012).

Emil Stefanov and Elaine Shi. 2013a. Multi-cloud oblivious storage. In *ACM Conference on Computer and Communications Security (CCS'13)*. ACM.

Emil Stefanov and Elaine Shi. 2013b. ObliviStore: High performance oblivious cloud storage. In *2013 IEEE Symposium on Security and Privacy (SP'13)*. IEEE, 253–267.

Emil Stefanov, Elaine Shi, and Dawn Song. 2012. Towards practical oblivious RAM. In *NDSS*.

Xiao Wang, Hubert Chan, and Elaine Shi. 2015. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 850–861.

Xiao Shaun Wang, Kartik Nayak, Chang Liu, T. H. Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 215–226.

Peter Williams and Radu Sion. 2008. Usable PIR. In *NDSS*.

Peter Williams and Radu Sion. 2012. Single round access privacy on outsourced storage. In *CCS*. ACM, 293–304.

Peter Williams, Radu Sion, and Bogdan Carbunar. 2008. Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In *CCS*. ACM, 139–148.

Peter Williams, Radu Sion, and Alin Tomescu. 2012. Privatefs: A parallel oblivious file system. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. ACM, 977–988.

Received April 2014; revised August 2017; accepted January 2018