Constants Count: Practical Improvements to Oblivious RAM

Ling Ren, Christopher Fletcher, and Albert Kwon, *Massachusetts Institute of Technology;* Emil Stefanov, *University of California, Berkeley;* Elaine Shi, *Cornell University;* Marten van Dijk, *University of Connecticut;* Srinivas Devadas, *Massachusetts Institute of Technology*

https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ren-ling

This paper is included in the Proceedings of the
24th USENIX Security Symposium

August 12–14, 2015 • Washington, D.C.

ISBN 978-1-931971-232

Constants Count: Practical Improvements to Oblivious RAM

Ling Ren
MIT

Christopher Fletcher
MIT

Albert Kwon
MIT

Emil Stefanov
UC Berkeley

Elaine Shi
Cornell University

Marten van Dijk
UConn

Srinivas Devadas
MIT

Abstract

Oblivious RAM (ORAM) is a cryptographic primitive that hides memory access patterns as seen by untrusted storage. This paper proposes Ring ORAM, the most bandwidth-efficient ORAM scheme for the small client storage setting in both theory and practice. Ring ORAM is the first tree-based ORAM whose bandwidth is independent of the ORAM bucket size, a property that unlocks multiple performance improvements. First, Ring ORAM's overall bandwidth is 2.3× to 4× better than Path ORAM, the prior-art scheme for small client storage. Second, if memory can perform simple untrusted computation, Ring ORAM achieves constant online bandwidth ($\sim 60\times$ improvement over Path ORAM for practical parameters). As a case study, we show Ring ORAM speeds up program completion time in a secure processor by 1.5× relative to Path ORAM. On the theory side, Ring ORAM features a tighter and significantly simpler analysis than Path ORAM.

1 Introduction

With cloud computing and storage gaining popularity, privacy of users' sensitive data has become a large concern. It is well known, however, that encryption alone is not enough to ensure data privacy. Even after encryption, a malicious server still learns a user's access pattern, e.g., how frequently each piece of data is accessed, if the user scans, binary searches or randomly accesses her data at different stages. Prior works have shown that access patterns can reveal a lot of information about encrypted files [14] or private user data in computation outsourcing [32, 18].

Oblivious RAM (ORAM) is a cryptographic primitive that *completely* eliminates the information leakage in memory access traces. In an ORAM scheme, a *client* (e.g., a local machine) accesses data blocks residing on a *server*, such that for any two logical access sequences of the same length, the observable communications between the client and the server are computationally indistinguishable.

ORAMs are traditionally evaluated by *bandwidth*—the number of blocks that have to be transferred between the client and the server to access one block, *client storage*—the amount of trusted local memory required at the client side, and *server storage*—the amount of untrusted memory required at the server side. All three metrics are measured as functions of N, the total number of data blocks in the ORAM.

A factor that determines which ORAM scheme to use is whether the client has a large (GigaBytes or larger) or small (KiloBytes to MegaBytes) storage budget. An example of large client storage setting is remote oblivious file servers [30, 17, 24, 3]. In this setting, a user runs on a local desktop machine and can use its main memory or disk for client storage. Given this large client storage budget, the preferred ORAM scheme to date is the SSS construction [25], which has about $1 \cdot \log N$ bandwidth and typically requires GigaBytes of client storage.

In the same file server application, however, if the user is instead on a mobile phone, the client storage will have to be small. A more dramatic example for small client storage is when the client is a remote secure processor — in which case client storage is restricted to the processor's scarce on-chip memory. Partly for this reason, all secure processor proposals [18, 16, 8, 31, 22, 7, 5, 6] have adopted Path ORAM [27] which allows for small (typically KiloBytes of) client storage.

The majority of this paper focuses on the small client storage setting and Path ORAM. In fact, our construction is an improvement to Path ORAM. However, in Section 7, we show that our techniques can be easily extended to obtain a competitive large client storage ORAM.

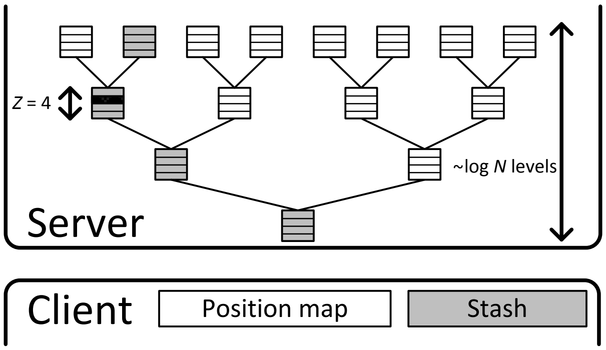

Figure 1: Path ORAM server and client storage. Suppose the **black** block is mapped to the shaded path. In that case, the block may reside in any slot along the path or in the stash (client storage).

1.1 Path ORAM and Challenges

We now give a brief overview of Path ORAM (for more details, see [27]). Path ORAM follows the tree-based ORAM paradigm [23] where server storage is structured as a binary tree of roughly $\log N$ levels. Each node in the tree is a bucket that can hold up to a small number Z of data blocks. Each path in the tree is defined as the sequence of buckets from the root of the tree to some leaf node. Each block is mapped to a random path, and must reside somewhere on that path. To access a block, the Path ORAM algorithm first looks up a position map, a table in client storage which tracks the path each block is currently mapped to, and then reads all the ($\sim Z \log N$) blocks on that path into a client-side data structure called the stash. The requested block is then remapped to a new random path and the position map is updated accordingly. Lastly, the algorithm invokes an eviction procedure which writes the same path we just read from, percolating blocks down that path. (Other tree-based ORAMs use different eviction algorithms that are less effective than Path ORAM, and hence the worse performance.)

The bandwidth of Path ORAM is $2Z \log N$ because each access reads and writes a path in the tree. To prevent blocks from accumulating in client storage, the bucket size Z has to be at least 4 (experimentally verified [27, 18]) or 5 (theoretically proven [26]).

We remind readers not to confuse the above read/write path operation with reading/writing data blocks. In ORAM, both reads and writes to a data block are served by the read path operation, which moves the requested block into client storage to be operated upon secretly. The sole purpose of the write path operation is to evict blocks from the stash and percolate blocks down the tree.

	Online Bandwidth	Overall Bandwidth
Path ORAM	$Z \log N = 4 \log N$	$2Z \log N = 8 \log N$
Ring ORAM	$\sim 1 \cdot \log N$	3-3.5$\log N$
Ring ORAM + XOR	~ 1	2-2.5$\log N$

Table 1: **Our contributions.** Overheads are relative to an insecure system. Ranges in constants for Ring ORAM are due to different parameter settings. The bandwidth cost of tree ORAM recursion [23, 26] is small ($< 3\%$) and thus excluded. XOR refers to the XOR technique from [3].

Despite being a huge improvement over prior schemes, Path ORAM is still plagued with several important challenges. First, the constant factor $2Z \geq 8$ is substantial, and brings Path ORAM's bandwidth overhead to $> 150\times$ for practical parameterizations. In contrast, the SSS construction does not have this bucket size parameter and can achieve close to $1 \cdot \log N$ bandwidth. (This bucket-size-dependent bandwidth is exactly why Path ORAM is dismissed in the large client storage setting.)

Second, despite the importance of overall bandwidth, online bandwidth—which determines response time—is equally, if not more, important in practice. For Path ORAM, half of the overall bandwidth must be incurred online. Again in contrast, an earlier work [3] reduced the SSS ORAM's online bandwidth to $O(1)$ by granting the server the ability to perform simple XOR computations. Unfortunately, their techniques do not apply to Path ORAM.

1.2 Our Contributions

In this paper, we propose Ring ORAM to address both challenges simultaneously. Our key technical achievement is to carefully re-design the tree-based ORAM such that the online bandwidth is $O(1)$, and the amortized overall bandwidth is independent of the bucket size. We compare bandwidth overhead with Path ORAM in Table 1. The major contributions of Ring ORAM include:

- **Small online bandwidth.** We provide the first tree-based ORAM scheme that achieves ~ 1 online bandwidth, relying only on very simple, untrusted computation logic on the server side. This represents at least $60\times$ improvement over Path ORAM for reasonable parameters.

- **Bucket-size independent overall bandwidth.** While all known tree-based ORAMs incur an overall bandwidth cost that depends on the bucket size, Ring ORAM eliminates this dependence, and improves overall bandwidth by $2.3\times$ to $4\times$ relative to Path ORAM.

- **Simple and tight theoretical analysis.** Using novel proof techniques based on Ring ORAM's eviction algorithm, we obtain a much simpler and tighter theoretical analysis than that of Path ORAM. Of independent interest, we note that the proof of Lemma 1 in [27], a crucial lemma for both Path ORAM and this paper, is incomplete (the lemma itself is correct). We give a rigorous proof for that lemma in this paper.

As mentioned, one main application of small client storage ORAM is for the secure processor setting. We simulate Ring ORAM in the secure processor setting and confirm that the improvement in bandwidth over Path ORAM translates to a 1.5× speedup in program completion time. Combined with all other known techniques, the average program slowdown from using an ORAM is 2.4× over a set of SPEC and database benchmarks.

Extension to larger client storage. Although our initial motivation was to design an optimized ORAM scheme under small client storage, as an interesting byproduct, Ring ORAM can be easily extended to achieve competitive performance in the large client storage setting. This makes Ring ORAM a good candidate in oblivious cloud storage, because as a tree-based ORAM, Ring ORAM is easier to analyze, implement and de-amortize than hierarchical ORAMs like SSS [25]. Therefore, Ring ORAM is essentially *a united paradigm for ORAM constructions in both large and small client storage settings.*

Organization. In the rest of this introduction, we give an overview of our techniques to improve ORAM's online and overall bandwidth. Section 2 gives a formal security definition for ORAM. Section 3 explains the Ring ORAM protocol in detail. Section 4 gives a complete formal analysis for bounding Ring ORAM's client storage. Section 5 analyzes Ring ORAM's bandwidth and gives a methodology for setting parameters optimally. Section 6 compares Ring ORAM to prior work in terms of bandwidth vs. client storage and performance in a secure processor setting. Section 7 describes how to extend Ring ORAM to the large client storage setting. Section 8 gives related work and Section 9 concludes.

1.3 Overview of Techniques

We now explain our key technical insights. At a high level, our scheme also follows the tree-based ORAM paradigm [23]. Server storage is a binary tree where each node (a bucket) contains up to Z blocks and blocks percolate down the tree during ORAM evictions. We introduce the following non-trivial techniques that allow us to achieve significant savings in both online and overall bandwidth costs.

Eliminating online bandwidth's dependence on bucket size. In Path ORAM, reading a block would amount to reading and writing all Z slots in all buckets on a path. Our first goal is to *read only one block from each bucket* on the path. To do this, we randomly permute each bucket and store the permutation in each bucket as additional metadata. Then, by reading only metadata, the client can determine whether the requested block is in the present bucket or not. If so, the client relies on the stored permutation to read the block of interest from its random offset. Otherwise, the client reads a "fresh" (unread) dummy block, also from a random offset. We stress that the metadata size is typically much smaller than the block size, so the cost of reading metadata can be ignored.

For the above approach to be secure, it is imperative that each block in a bucket should be read at most once—a key idea also adopted by Goldreich and Ostrovsky in their early ORAM constructions [11]. Notice that any real block is naturally read only once, since once a real block is read, it will be invalidated from the present bucket, and relocated somewhere else in the ORAM tree. But dummy blocks in a bucket can be exhausted if the bucket is read many times. When this happens (which is public information), Ring ORAM introduces an *early reshuffle* procedure to reshuffle the buckets that have been read too many times. Specifically, suppose that each bucket is guaranteed to have S dummy blocks, then a bucket must be reshuffled every S times it is read.

We note that the above technique also gives an additional nice property: out of the $O(\log N)$ blocks the client reads, only 1 of them is a real block (i.e., the block of interest); all the others are dummy blocks. If we allow some simple computation on the memory side, we can immediately apply the XOR trick from Burst ORAM [3] to get $O(1)$ online bandwidth. In the XOR trick, the server simply XORs these encrypted blocks and sends a single, XOR'ed block to the client. The client can reconstruct the ciphertext of all the dummy blocks, and XOR them away to get back the encrypted real block.

Eliminating overall bandwidth's dependence on bucket size. Unfortunately, naïvely applying the above strategy will dramatically increase offline and overall bandwidth. The more dummy slots we reserve in each bucket (i.e., a large S), the more expensive ORAM evictions become, since they have to read and write all the blocks in a bucket. But if we reserve too few dummy slots, we will frequently run out of dummy blocks and have to call *early reshuffle*, also increasing overall bandwidth.

We solve the above problem with several additional techniques. First, we design a new eviction procedure that improves eviction quality. At a high level, Ring

ORAM performs evictions on a path in a similar fashion as Path ORAM, but eviction paths are selected based on a reverse lexicographical order [9], which evenly spreads eviction paths over the entire tree. The improved eviction quality allows us to perform evictions less frequently, only once every A ORAM accesses, where A is a new parameter. We then develop a proof that crucially shows A can approach $2Z$ while still ensuring negligible ORAM failure probability. The proof may be of independent interest as it uses novel proof techniques and is significantly simpler than Path ORAM's proof. The **amortized offline bandwidth** is now roughly $\frac{2Z}{A}\log N$, which **does not depend on the bucket size Z either**.

Second, bucket reshuffles can naturally piggyback on ORAM evictions. The balanced eviction order further ensures that every bucket will be reshuffled regularly. Therefore, we can set the reserved dummy slots S in accordance with the eviction frequency A, such that early reshuffles contribute little ($< 3\%$) to the overall bandwidth.

Putting it all Together. None of the aforementioned ideas would work alone. Our final product, Ring ORAM, stems from intricately combining these ideas in a nontrivial manner. For example, observe how our two main techniques act like two sides of a lever: (1) permuted buckets such that only 1 block is read per bucket; and (2) high quality and hence less frequent evictions. While permuted buckets make reads cheaper, they require adding dummy slots and would dramatically increase eviction overhead without the second technique. At the same time, less frequent evictions require increasing bucket size Z; without permuted buckets, ORAM reads blow up and nullify any saving on evictions. Additional techniques are needed to complete the construction. For example, early reshuffles keep the number of dummy slots small; piggyback reshuffles and load-balancing evictions keep the early reshuffle rate low. Without all of the above techniques, one can hardly get any improvement.

2 Security Definition

We adopt the standard ORAM security definition. Informally, the server should not learn anything about: 1) which data the client is accessing; 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. Like previous work, we do not consider information leakage through the timing channel, such as when or how frequently the client makes data requests.

Notation	Meaning
N	Number of real data blocks in ORAM
L	Depth of the ORAM tree
Z	Maximum number of real blocks per bucket
S	Number of slots reserved for dummies per bucket
B	Data block size (in bits)
A	Eviction rate (larger means less frequent)
$\mathcal{P}(l)$	Path l
$\mathcal{P}(l,i)$	The i-th bucket (towards the root) on $\mathcal{P}(l)$
$\mathcal{P}(l,i,j)$	The j-th slot in bucket $\mathcal{P}(l,i)$

Table 2: ORAM parameters and notations.

Definition 1. (ORAM Definition) *Let*

$$\overleftarrow{y} = ((\mathsf{op}_M, \mathsf{addr}_M, \mathsf{data}_M), \ldots, (\mathsf{op}_1, \mathsf{addr}_1, \mathsf{data}_1))$$

denote a data sequence of length M, where op_i *denotes whether the i-th operation is a read or a write,* addr_i *denotes the address for that access and* data_i *denotes the data (if a write). Let* $\mathsf{ORAM}(\overleftarrow{y})$ *be the resulting sequence of operations between the client and server under an ORAM algorithm. The ORAM protocol guarantees that for any* $\overleftarrow{y}$ *and* $\overleftarrow{y}'$*,* $\mathsf{ORAM}(\overleftarrow{y})$ *and* $\mathsf{ORAM}(\overleftarrow{y}')$ *are computationally indistinguishable if* $|\overleftarrow{y}| = |\overleftarrow{y}'|$*, and also that for any* $\overleftarrow{y}$ *the data returned to the client by ORAM is consistent with* $\overleftarrow{y}$ *(i.e., the ORAM behaves like a valid RAM) with overwhelming probability.*

We remark that for the server to perform computations on data blocks [3], $\mathsf{ORAM}(\overleftarrow{y})$ and $\mathsf{ORAM}(\overleftarrow{y}')$ include those operations. To satisfy the above security definition, it is implied that these operations also cannot leak any information about the access pattern.

3 Ring ORAM Protocol

3.1 Overview

We first describe Ring ORAM in terms of its server and client data structures. All notation used throughout the rest of the paper is summarized in Table 2.

Server storage is organized as a binary tree of buckets where each bucket has a small number of slots to hold blocks. Levels in the tree are numbered from 0 (the root) to L (inclusive, the leaves) where $L = O(\log N)$ and N is the number of blocks in the ORAM. Each bucket has $Z + S$ slots and a small amount of metadata. Of these slots, up to Z slots may contain real blocks and the remaining S slots are reserved for dummy blocks as described in Section 1.3. Our theoretical analysis in Section 4 will show that to store N blocks in Ring ORAM, the physical ORAM tree needs roughly $6N$ to $8N$ slots. Experiments show that server storage in practice for both Ring ORAM and Path ORAM can be $2N$ or even smaller.

Client storage is made up of a position map and a stash. The position map is a dictionary that maps each block in the ORAM to a random leaf in the ORAM tree (each leaf is given a unique identifier). The stash buffers blocks that have not been evicted to the ORAM tree and additionally stores $Z(L+1)$ blocks on the eviction path during an eviction operation. We will prove in Section 4 that stash overflow probability decreases exponentially as stash capacity increases, which means our required stash size is the same as Path ORAM. The position map stores $N * L$ bits, but can be squashed to constant storage using the standard recursion technique (Section 3.7).

Main invariants. Ring ORAM has two main invariants:

1. (Same as Path ORAM): Every block is mapped to a leaf chosen uniformly at random in the ORAM tree. If a block a is mapped to leaf l, block a is contained either in the stash or in some bucket along the path from the root of the tree to leaf l.
2. **(Permuted buckets)** For every bucket in the tree, the physical positions of the $Z+S$ dummy and real blocks in each bucket are randomly permuted with respect to all past and future writes to that bucket.

Since a leaf uniquely determines a path in a binary tree, we will use leaves/paths interchangeably when the context is clear, and denote path l as $\mathcal{P}(l)$.

Access and Eviction Operations. The Ring ORAM access protocol is shown in Algorithm 1. Each access is broken into the following four steps:

1.) Position Map lookup (Lines 3-5): Look up the position map to learn which path l the block being accessed is currently mapped to. Remap that block to a new random path l'.

This first step is identical to other tree-based ORAMs [23, 27]. But the rest of the protocol differs substantially from previous tree-based schemes, and we highlight our key innovations in **bold**.

2.) Read Path (Lines 6-15): The ReadPath(l,a) operation reads all buckets along $\mathcal{P}(l)$ to look for the block of interest (block a), and then reads that block into the stash. The block of interest is then updated in stash on a write, or is returned to the client on a read. We remind readers again that both reading and writing a data block are served by a ReadPath operation.

Unlike prior tree-based schemes, our ReadPath operation **only reads one block from each bucket—the block of interest if found or a previously-unread dummy block otherwise.** This is safe because of Invariant 2, above: each bucket is permuted randomly, so the slot being read looks random to an observer. This lowers the bandwidth overhead of ReadPath (i.e., online bandwidth) to $L+1$ blocks (the number of levels in the tree) or even a single block if the XOR trick is applied (Section 3.2).

Algorithm 1 Non-recursive Ring ORAM.

```
1:  function ACCESS($a$, op, data$'$)
2:      Global/persistent variables: round

3:      $l' \leftarrow$ UniformRandom$(0, 2^L - 1)$
4:      $l \leftarrow$ PositionMap$[a]$
5:      PositionMap$[a] \leftarrow l'$

6:      data $\leftarrow$ ReadPath$(l, a)$
7:      if data $= \perp$ then
8:          ▷ If block $a$ is not found on path $l$, it must
9:          be in Stash ◁
10:         data $\leftarrow$ read and remove $a$ from Stash
11:     if op $=$ read then
12:         return data to client
13:     if op $=$ write then
14:         data $\leftarrow$ data$'$
15:     Stash $\leftarrow$ Stash $\cup (a, l', \text{data})$

16:     round $\leftarrow$ round $+ 1 \mod A$
17:     if round $\overset{?}{=} 0$ then
18:         EvictPath()

19:     EarlyReshuffle$(l)$
```

3.) Evict Path (Line 16-18): The EvictPath operation reads Z blocks (all the remaining real blocks, and potentially some dummy blocks) from each bucket along a path into the stash, and then fills that path with blocks from the stash, trying to push blocks as far down towards the leaves as possible. The sole purpose of an eviction operation is to push blocks back to the ORAM tree to keep the stash occupancy low.

Unlike Path ORAM, eviction in Ring ORAM **selects paths in the reverse lexicographical order, and does not happen on every access. Its rate is controlled by a public parameter A: every A ReadPath operations trigger a single EvictPath operation.** This means Ring ORAM needs much fewer eviction operations than Path ORAM. We will theoretically derive a tight relationship between A and Z in Section 4.

4.) Early Reshuffles (Line 19): Finally, we perform **a maintenance task called** EarlyReshuffle **on $\mathcal{P}(l)$, the path accessed by** ReadPath**.** This step is crucial in maintaining blocks randomly shuffled in each bucket, which enables ReadPath to securely read only one block from each bucket.

We will present details of ReadPath, EvictPath and EarlyReshuffle in the next three subsections. We defer low-level details for helper functions needed in these three subroutines to Appendix A. We explain the security for each subroutine in Section 3.5. Finally, we discuss additional optimizations in Section 3.6 and recursion in Section 3.7.

3.2 Read Path Operation

Algorithm 2 ReadPath procedure.

```
1: function ReadPath($l, a$)
2:     data ← $\perp$
3:     for $i \leftarrow 0$ to $L$ do
4:         offset ← GetBlockOffset($\mathcal{P}(l,i), a$)
5:         data′ ← $\mathcal{P}(l,i,$offset$)$
6:         Invalidate $\mathcal{P}(l,i,$offset$)$
7:         if data′ $\neq \perp$ then
8:             data ← data′
9:         $\mathcal{P}(l,i)$.count ← $\mathcal{P}(l,i)$.count + 1
       return data
```

The ReadPath operation is shown in Algorithm 2. For each bucket along the current path, ReadPath selects a *single* block to read from that bucket. For a given bucket, if the block of interest lives in that bucket, we read and invalidate the block of interest. Otherwise, we read and invalidate a randomly-chosen dummy block that is still valid at that point. The index of the block to read (either real or random) is returned by the GetBlockOffset function whose detailed description is given in Appendix A.

Reading a single block per bucket is crucial for our bandwidth improvements. In addition to reducing online bandwidth by a factor of Z, it allows us to use larger Z and A to decrease overall bandwidth (Section 5). Without this, read bandwidth is proportional to Z, and the cost of larger Z on reads outweighs the benefits.

Bucket Metadata. Because the position map only tracks the *path* containing the block of interest, the client does not know where in each bucket to look for the block of interest. Thus, for each bucket we must store the permutation in the bucket metadata that maps each real block in the bucket to one of the $Z+S$ slots (Lines 4, GetBlockOffset) as well as some additional metadata. Once we know the offset into the bucket, Line 5 reads the block in the slot, and invalidates it. We describe all metadata in Appendix A, but make the important point that the metadata is small and independent of the block size.

One important piece of metadata to mention now is a counter which tracks how many times it has been read since its last eviction (Line 9). If a bucket is read too many (S) times, it may run out of dummy blocks (i.e., all the dummy blocks have been invalidated). On future accesses, if additional dummy blocks are requested from this bucket, we cannot re-read a previously invalidated dummy block: doing so reveals to the adversary that the block of interest is not in this bucket. Therefore, we need to reshuffle single buckets on-demand as soon as they are touched more than S times using EarlyReshuffle (Section 3.4).

XOR Technique. We further make the following key observation: during our ReadPath operation, each block returned to the client is a dummy block except for the block of interest. This means our scheme can also take advantage of the XOR technique introduced in [3] to reduce online bandwidth overhead to $O(1)$. To be more concrete, on each access ReadPath returns $L+1$ blocks in ciphertext, one from each bucket, $\mathsf{Enc}(b_0, r_0), \mathsf{Enc}(b_2, r_2), \cdots, \mathsf{Enc}(b_L, r_L)$. Enc is a randomized symmetric scheme such as AES counter mode with nonce r_i. With the XOR technique, ReadPath will return *a single ciphertext* — the ciphertext of all the blocks XORed together, namely $\mathsf{Enc}(b_0, r_0) \oplus \mathsf{Enc}(b_2, r_2) \oplus \cdots \oplus \mathsf{Enc}(b_L, r_L)$. The client can recover the encrypted block of interest by XORing the returned ciphertext with the encryptions of all the dummy blocks. To make computing each dummy block's encryption easy, the client can set the plaintext of all dummy blocks to a fixed value of its choosing (e.g., 0).

3.3 Evict Path Operation

Algorithm 3 EvictPath procedure.

```
function EvictPath
    Global/persistent variables $G$ initialized to 0
    $l \leftarrow G \mod 2^L$
    $G \leftarrow G + 1$
    for $i \leftarrow 0$ to $L$ do
        Stash ← Stash ∪ ReadBucket($\mathcal{P}(l,i)$)
    for $i \leftarrow L$ to 0 do
        WriteBucket($\mathcal{P}(l,i)$, Stash)
        $\mathcal{P}(l,i)$.count ← 0
```

The EvictPath routine is shown in Algorithm 3. As mentioned, evictions are scheduled statically: one evic-

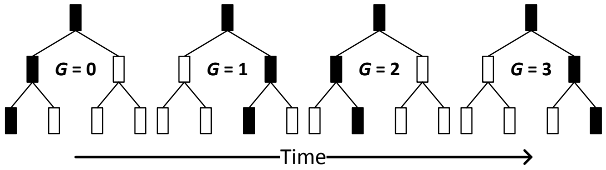

Figure 2: Reverse-lexicographic order of paths used by EvictPath. After path $G = 3$ is evicted to, the order repeats.

tion operation happens after every A reads. At a high level, an eviction operation reads all remaining real blocks on a path (in a secure fashion), and tries to push them down that path as far as possible. The leaf-to-root order in the writeback step (Lines 7) reflects that we wish to fill the deepest buckets as fully as possible. (For readers who are familiar with Path ORAM, EvictPath is like a Path ORAM access where no block is accessed and therefore no block is remapped to a new leaf.)

We emphasize two unique features of Ring ORAM eviction operations. First, evictions in Ring ORAM are performed to paths in a specific order called the *reverse-lexicographic order*, first proposed by Gentry et al. [9] and shown in Figure 2. The reverse-lexicographic order eviction aims to minimize the overlap between consecutive eviction paths, because (intuitively) evictions to the same bucket in consecutive accesses are less useful. This improves eviction quality and allows us to reduce the frequency of eviction. Evicting using this static order is also a key component in simplifying our theoretical analysis in Section 4.

Second, buckets in Ring ORAM need to be randomly shuffled (Invariant 2), and we mostly rely on EvictPath operations to keep them shuffled. An EvictPath operation reads Z blocks from each bucket on a path into the stash, and writes out $Z+S$ blocks (only up to Z are real blocks) to each bucket, randomly permuted. The details of reading/writing buckets (ReadBucket and WriteBucket) are deferred to Appendix A.

3.4 Early Reshuffle Operation

Algorithm 4 EarlyReshuffle procedure.

```
function EarlyReshuffle($l$)
    for $i \leftarrow 0$ to $L$ do
        if $\mathcal{P}(l,i)$.count $\geq S$ then
            Stash $\leftarrow$ Stash $\cup$ ReadBucket($\mathcal{P}(l,i)$)
            WriteBucket($\mathcal{P}(l,i)$, Stash)
            $\mathcal{P}(l,i)$.count $\leftarrow 0$
```

Due to randomness, a bucket can be touched $> S$ times by ReadPath operations before it is reshuffled by the scheduled EvictPath. If this happens, we call EarlyReshuffle on that bucket to reshuffle it before the bucket is read again (see Section 3.2). More precisely, after each ORAM access EarlyReshuffle goes over all the buckets on the read path, and reshuffles all the buckets that have been accessed more than S times by performing ReadBucket and WriteBucket. ReadBucket and WriteBucket are the same as in EvictPath: that is, ReadBucket reads exactly Z slots in the bucket and WriteBucket re-permutes and writes back $Z+S$ real/dummy blocks. We note that though S does not affect security (Section 3.5), it clearly has an impact on performance (how often we shuffle, the extra cost per reshuffle, etc.). We discuss how to optimally select S in Section 5.

3.5 Security Analysis

Claim 1. ReadPath *leaks no information.*

The path selected for reading will look random to any adversary due to Invariant 1 (leaves are chosen uniformly at random). From Invariant 2, we know that every bucket is randomly shuffled. Moreover, because we invalidate any block we read, we will never read the same slot. Thus, any sequence of reads (real or dummy) to a bucket between two shuffles is indistinguishable. Thus the adversary learns nothing during ReadPath. □

Claim 2. EvictPath *leaks no information.*

The path selected for eviction is chosen statically, and is public (reverse-lexicographic order). ReadBucket always reads exactly Z blocks from random slots. WriteBucket similarly writes $Z+S$ encrypted blocks in a data-independent fashion. □

Claim 3. EarlyShuffle *leaks no information.*

To which buckets EarlyShuffle operations occur is publicly known: the adversary knows how many times a bucket has been accessed since the last EvictPath to that bucket. ReadBucket and WriteBucket are secure as per observations in Claim 2. □

The three subroutines of the Ring ORAM algorithm are the only operations that cause externally observable behaviors. Claims 1, 2, and 3 show that the subroutines are secure. We have so far assumed that path remapping and bucket permutation are truly random, which gives unconditional security. If pseudorandom numbers are used instead, we have computational security through similar arguments.

3.6 Other Optimizations

Minimizing roundtrips. To keep the presentation simple, we wrote the ReadPath (EvictPath) algorithms to process buckets one by one. In fact, they can be performed for all buckets on the path in parallel which reduces the number of roundtrips to 2 (one for metadata and one for data blocks).

Tree-top caching. The idea of tree-top caching [18] is simple: we can reduce the bandwidth for ReadPath and EvictPath by storing the top t (a new parameter) levels of the Ring ORAM tree at the client as an extension of the stash[1]. For a given t, the stash grows by approximately $2^t Z$ blocks.

De-amortization. We can de-amortize the expensive EvictPath operation through a period of A accesses, simply by reading/writing a small number of blocks on the eviction path after each access. After de-amortization, worst-case overall bandwidth equals average overall bandwidth.

3.7 Recursive Construction

With the construction given thus far, the client needs to store a large position map. To achieve small client storage, we follow the standard recursion idea in tree-based ORAMs [23]: instead of storing the position map on the client, we store the position map on a smaller ORAM on the server, and store only the position map for the smaller ORAM. The client can recurse until the final position map becomes small enough to fit in its storage. For reasonably block sizes (e.g., 4 KB), recursion contributes very little to overall bandwidth (e.g., $< 5\%$ for a 1 TB ORAM) because the position map ORAMs use much smaller blocks [26]. Since recursion for Ring ORAM behaves in the same way as all the other tree-based ORAMs, we omit the details.

4 Stash Analysis

In this section we analyze the stash occupancy for a non-recursive Ring ORAM. Following the notations in Path ORAM [27], by $\mathsf{ORAM}_L^{Z,A}$ we denote a non-recursive Ring ORAM with $L+1$ levels, bucket size Z and one eviction per A accesses. The root is at level 0 and the leaves are at level L. We define the stash occupancy $\mathsf{st}(\mathcal{S}_Z)$ to be the number of real blocks in the stash after a sequence of ORAM sequences (this notation will be further explained later). We will prove that $\Pr[\mathsf{st}(\mathcal{S}_Z) > R]$ decreases exponentially in R for certain Z and A combinations. As it turns out, the deterministic eviction pattern in Ring ORAM dramatically simplifies the proof.

We note here that the reshuffling of a bucket does not affect the occupancy of the bucket, and is thus irrelevant to the proof we present here.

4.1 Proof outline

The proof consists of the two steps. The first step is the same as Path ORAM, and needs Lemma 1 and Lemma 2 in the Path ORAM paper [27], which we restate in Section 4.2. We introduce ∞-ORAM, which has an infinite bucket size and after a post-processing step has exactly the same distribution of blocks over all buckets and the stash (Lemma 1). Lemma 2 says the stash occupancy of ∞-ORAM after post-processing is greater than R if and only if there exists a subtree T in ∞-ORAM whose "occupancy" exceeds its "capacity" by more than R. We note, however, that the Path ORAM [27] paper only gave intuition for the proof of Lemma 1, and unfortunately did not capture of all the subtleties. We will rigorously prove that lemma, which turns out to be quite tricky and requires significant changes to the post-processing algorithm.

The second step (Section 4.3) is much simpler than the rest of Path ORAM's proof, thanks to Ring ORAM's static eviction pattern. We simply need to calculate the expected occupancy of subtrees in ∞-ORAM, and apply a Chernoff-like bound on their actual occupancy to complete the proof. We do not need the complicated eviction game, negative association, stochastic dominance, etc., as in the Path ORAM proof [26].

For readability, we will defer the proofs of all lemmas to Appendix B.

4.2 ∞-ORAM

We first introduce ∞-ORAM, denoted as $\mathsf{ORAM}_L^{\infty,A}$. Its buckets have infinite capacity. It receives the same input request sequence as $\mathsf{ORAM}_L^{Z,A}$. We then label buckets linearly such that the two children of bucket b_i are b_{2i} and b_{2i+1}, with the root bucket being b_1. We define the stash to be b_0. We refer to b_i of $\mathsf{ORAM}_L^{\infty,A}$ as b_i^∞, and b_i of $\mathsf{ORAM}_L^{Z,A}$ as b_i^Z. We further define ORAM *state*, which consists of the states of all the buckets in the ORAM, i.e., the blocks contained by each bucket. Let $\mathcal{S}_\infty$ be the state of $\mathsf{ORAM}_L^{\infty,A}$ and $\mathcal{S}_Z$ be the state of $\mathsf{ORAM}_L^{Z,A}$.

We now propose a new greedy post-processing algorithm G (different from the one in [27]), which by reassigning blocks in buckets makes each bucket b_i^∞ in ∞-ORAM contain the same set of blocks as b_i^Z. Formally, G takes as input $\mathcal{S}_\infty$ and $\mathcal{S}_Z$ after the same access sequence with the same randomness. For i from $2^{L+1}-1$ down to

[1] We call this optimization tree-top caching following prior work. But the word cache is a misnomer: the top t levels of the tree are *permanently* stored by the client.

1 (note that the decreasing order ensures that a parent is always processed later than its children), G processes the blocks in bucket b_i^∞ in the following way:

1. For those blocks that are also in b_i^Z, keep them in b_i^∞.
2. For those blocks that are not in b_i^Z but in some ancestors of b_i^Z, move them from b_i^∞ to $b_{i/2}^\infty$ (the parent of b_i^∞, and note that the division includes flooring). If such blocks exist and the number of blocks remaining in b_i^∞ is less than Z, raise an error.
3. If there exists a block in b_i^∞ that is in neither b_i^Z nor any ancestor of b_i^Z, raise an error.

We say $G_{\mathcal{S}_Z}(\mathcal{S}_\infty) = \mathcal{S}_Z$, if no error occurs during G and b_i^∞ after G contains the same set of blocks as b_i^Z for $i = 0, 1, \cdots 2^{L+1}$.

Lemma 1. *$G_{\mathcal{S}_Z}(\mathcal{S}_\infty) = \mathcal{S}_Z$ after the same ORAM access sequence with the same randomness.*

Next, we investigate what state $\mathcal{S}_\infty$ will lead to the stash occupancy of more than R blocks in a post-processed ∞-ORAM. We say a subtree T is a rooted subtree, denoted as $T \in \mathsf{ORAM}_L^{\infty,A}$ if T contains the root of $\mathsf{ORAM}_L^{\infty,A}$. This means that if a node in $\mathsf{ORAM}_L^{\infty,A}$ is in T, then so are all its ancestors. We define $n(T)$ to be the total number of nodes in T. We define $c(T)$ (the capacity of T) to be the maximum number of blocks T can hold; for Ring ORAM $c(T) = n(T) \cdot Z$. Lastly, we define $X(T)$ (the occupancy of T) to be the actual number of real blocks that are stored in T. The following lemma characterizes the stash size of a post-processed ∞-ORAM:

Lemma 2. *$\mathsf{st}(G_{\mathcal{S}_Z}(\mathcal{S}_\infty)) > R$ if and only if $\exists T \in \mathsf{ORAM}_L^{\infty,A}$ s.t. $X(T) > c(T) + R$ before post-processing.*

By Lemma 1 and Lemma 2, we have

$$\begin{aligned}
&\Pr[\mathsf{st}(\mathcal{S}_Z) > R] = \Pr[\mathsf{st}(G_{\mathcal{S}_Z}(\mathcal{S}_\infty)) > R] \\
&\le \sum_{T \in \mathsf{ORAM}_L^{\infty,A}} \Pr[X(T) > c(T) + R] \\
&< \sum_{n \ge 1} 4^n \max_{T : n(T) = n} \Pr[X(T) > c(T) + R] \qquad (1)
\end{aligned}$$

The above inequalities used a union bound and a bound on Catalan sequences.

4.3 Bounding the Stash Size

We first give a bound on the expected bucket load:

Lemma 3. *For any rooted subtree T in $\mathsf{ORAM}_L^{\infty,A}$, if the number of distinct blocks in the ORAM $N \le A \cdot 2^{L-1}$, the expected load of T has the following upper bound:*

$$\forall T \in \mathsf{ORAM}_L^{\infty,A}, E[X(T)] \le n(T) \cdot A/2.$$

Let $X(T) = \sum_i X_i(T)$, where each $X_i(T) \in \{0,1\}$ and indicates whether the i-th block (can be either real or stale) is in T. Let $p_i = \Pr[X_i(T) = 1]$. $X_i(T)$ is completely determined by its time stamp i and the leaf label assigned to block i, so they are independent from each other (refer to the proof of Lemma 3). Thus, we can apply a Chernoff-like bound to get an exponentially decreasing bound on the tail distribution. To do so, we first establish a bound on $E\left[e^{tX(T)}\right]$ where $t > 0$,

$$\begin{aligned}
E\left[e^{tX(T)}\right] &= E\left[e^{t\sum_i X_i(T)}\right] = E\left[\Pi_i e^{tX_i(T)}\right] \\
&= \Pi_i E\left[e^{tX_i(T)}\right] \qquad \text{(by independence)} \\
&= \Pi_i \left(p_i(e^t - 1) + 1\right) \\
&\le \Pi_i \left(e^{p_i(e^t-1)}\right) = e^{(e^t-1)\Sigma_i p_i} \\
&= e^{(e^t-1)E[X(T)]} \qquad (2)
\end{aligned}$$

For simplicity, we write $n = n(T)$ and $a = A/2$. By Lemma 3, $E[X(T)] \le n \cdot a$. By the Markov Inequality, we have for all $t > 0$,

$$\begin{aligned}
\Pr[X(T) > c(T) + R] &= \Pr\left[e^{tX(T)} > e^{t(nZ+R)}\right] \\
&\le E\left[e^{tX(T)}\right] \cdot e^{-t(nZ+R)} \\
&\le e^{(e^t-1)an} \cdot e^{-t(nZ+R)} \\
&= e^{-tR} \cdot e^{-n[tZ - a(e^t-1)]}
\end{aligned}$$

Let $t = \ln(Z/a)$,

$$\Pr[X(T) > c(T) + R] \le (a/Z)^R \cdot e^{-n[Z\ln(Z/a) + a - Z]} \qquad (3)$$

Now we will choose Z and A such that $Z > a$ and $q = Z\ln(Z/a) + a - Z - \ln 4 > 0$. If these two conditions hold, from Equation (1) we have $t = \ln(Z/a) > 0$ and that the stash overflow probability decreases exponentially in the stash size R:

$$\Pr[\mathsf{st}(\mathcal{S}_Z) > R] \le \sum_{n \ge 1} (a/Z)^R \cdot e^{-qn} < \frac{(a/Z)^R}{1 - e^{-q}}.$$

4.4 Stash Size in Practice

Now that we have established that $Z\ln(2Z/A) + A/2 - Z - \ln 4 > 0$ ensures an exponentially decreasing stash overflow probability, we would like to know how tight this requirement is and what the stash size should be in practice.

We simulate Ring ORAM with $L = 20$ for over 1 Billion accesses in a random access pattern, and measure the stash occupancy (excluding the transient storage of a path). For several Z values, we look for the maximum A that results in an exponentially decreasing stash overflow

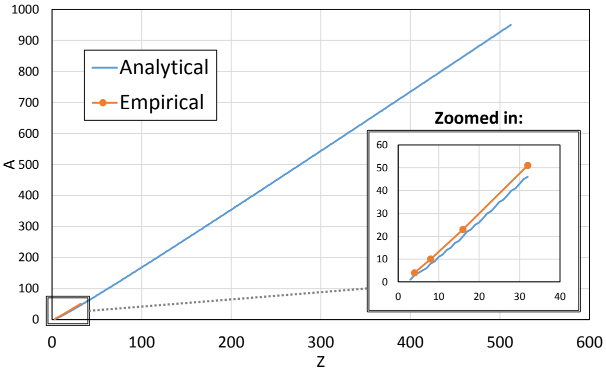

Figure 3: For each Z, determine analytically and empirically the maximum A that results in an exponentially decreasing stash failure probability.

		Z,A Parameters				
		4,3	8,8	16,20	32,46	16,23
		Max Stash Size				
λ	80	32	41	65	113	197
	128	51	62	93	155	302
	256	103	120	171	272	595

Table 3: Maximum stash occupancy for realistic security parameters (stash overflow probability $2^{-\lambda}$) and several choices of A and Z. $A = 23$ is the maximum achievable A for $Z = 16$ according to simulation.

probability. In Figure 3, we plot both the empirical curve based on simulation and the theoretical curve based on the proof. In all cases, the theoretical curve indicates a only slightly smaller A than we are able to achieve in simulation, indicating that our analysis is tight.

To determine required stash size in practice, Table 3 shows the extrapolated required stash size for a stash overflow probability of $2^{-\lambda}$ for several realistic λ. We show $Z = 16, A = 23$ for completeness: this is an aggressive setting that works for $Z = 16$ according to simulation but does not satisfy the theoretical analysis; observe that this point requires roughly 3× the stash occupancy for a given λ.

5 Bandwidth Analysis

In this section, we answer an important question: how do Z (the maximum number of real blocks per bucket), A (the eviction rate) and S (the number of extra dummies per bucket) impact Ring ORAM's performance (bandwidth)? By the end of the section, we will have a theoretically-backed analytic model that, given Z, selects optimal A and S to minimize bandwidth.

We first state an intuitive trade-off: for a given Z, increasing A causes stash occupancy to increase and bandwidth overhead to decrease. Let us first ignore early reshuffles and the XOR technique. Then, the overall bandwidth of Ring ORAM consists of ReadPath and EvictPath. ReadPath transfers $L+1$ blocks, one from each bucket. EvictPath reads Z blocks per bucket and writes $Z+S$ blocks per bucket, $(2Z+S)(L+1)$ blocks in total, but happens every A accesses. From the requirement of Lemma 3, we have $L = \log(2N/A)$, so the ideal amortized overall bandwidth of Ring ORAM is $(1+(2Z+S)/A)\log(4N/A)$. Clearly, a larger A improves bandwidth for a given Z as it reduces both eviction frequency and tree depth L. So we simply choose the largest A that satisfies the requirement from the stash analysis in Section 4.3.

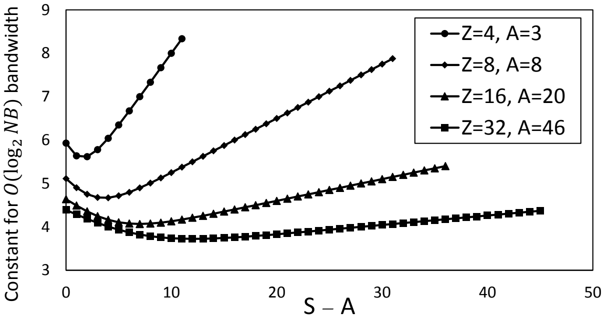

Figure 4: For different Z, and the corresponding optimal A, vary S and plot bandwidth overhead. We only consider $S \geq A$

Now we consider the extra overhead from early reshuffles. We have the following trade-off in choosing S: as S increases, the early reshuffle rate decreases (since we have more dummies per bucket) but the cost to read+write buckets during an EvictPath and EarlyReshuffle increases. This effect is shown in Figure 4 through simulation: for S too small, early shuffle rate is high and bandwidth increases; for S too large, eviction bandwidth dominates.

To analytically choose a good S, we analyze the early reshuffle rate. First, notice a bucket at level l in the Ring ORAM tree will be processed by EvictPath *exactly* once for every $2^l A$ ReadPath operations, due to the reverse-lexicographic order of eviction paths (Section 3.3). Second, each ReadPath operation is to an independent and uniformly random path and thus will touch any bucket in level l with equal probability of 2^{-l}. Thus, the distribution on the expected number of times ReadPath operations touch a given bucket in level l, between two consecutive EvictPath calls, is given by a binomial distribution of $2^l A$ trials and success probability 2^{-l}. The probability that a bucket needs to be early reshuffled before an EvictPath is given by a binomial distribution cumula-

Find largest $A \leq 2Z$ such that

$Z \ln(2Z/A) + A/2 - Z - \ln 4 > 0$ holds.

Find $S \geq 0$ that minimizes

$(2Z+S)(1+\mathsf{Poiss\_cdf}(S,A))$

Ring ORAM offline bandwidth is

$$\frac{(2Z+S)(1+\mathsf{Poiss\_cdf}(S,A))}{A} \cdot \log(4N/A)$$

Table 4: Analytic model for choosing parameters, given Z.

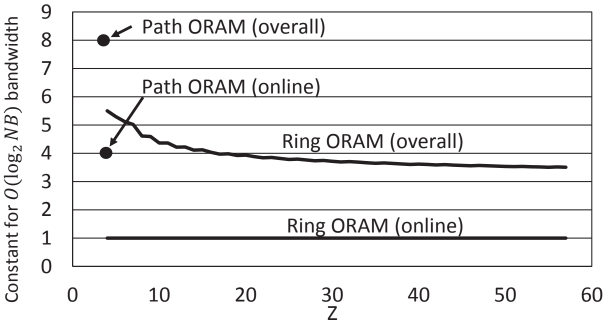

Figure 5: Overall bandwidth as a function of Z. Kinks are present in the graph because we always round A to the nearest integer. For Path ORAM, we only study $Z = 4$ since a larger Z strictly hurts bandwidth.

tive density function $\mathsf{Binom\_cdf}(S, 2^l A, 2^{-l})$.[2] Based on this analysis, the expected number of times any bucket is involved in ReadPath operations between consecutive EvictPath operations is A. Thus, we will only consider $S \geq A$ as shown in Figure 4 ($S < A$ is clearly bad as it needs too much early reshuffling).

We remark that the binomial distribution quickly converges to a Poisson distribution. So the amortized overall bandwidth, taking early reshuffles into account, can be accurately approximated as $(L+1)+(L+1)(2Z+S)/A \cdot (1+\mathsf{Poiss\_cdf}(S,A))$. We should then choose the S that minimizes the above formula. This method always finds the optimal S and perfectly matches the overall bandwidth in our simulation in Figure 4.

We recap how to choose A and S for a given Z in Table 4. For the rest of the paper, we will choose A and S this way unless otherwise stated. Using this method to set A and S, we show online and overall bandwidth as a function of Z in Figure 5. In the figure, Ring ORAM does not use the XOR technique on reads. For $Z = 50$, we achieve $\sim 3.5 \log N$ bandwidth; for very large Z, bandwidth approaches $3 \log N$. Applying the XOR technique, online bandwidth overhead drops to close to 1 which reduces overall bandwidth to $\sim 2.5 \log N$ for $Z = 50$ and $2 \log N$ for very large Z.

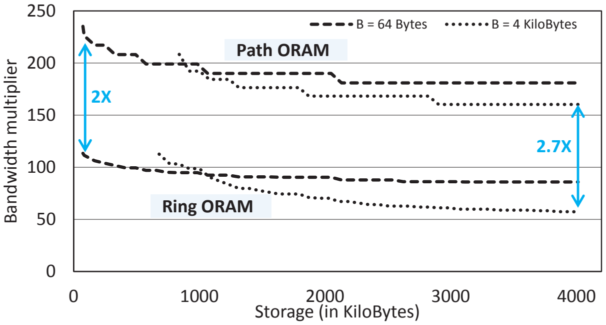

Figure 6: Bandwidth overhead vs. data block storage for 1 TB ORAM capacities and ORAM failure probability 2^{-80}.

6 Evaluation

6.1 Bandwidth vs. Client Storage

To give a holistic comparison between schemes, Figure 6 shows the best achievable bandwidth, for different client storage budgets, for Path ORAM and Ring ORAM. For each scheme in the figure, we apply all known optimizations and tune parameters to minimize overall bandwidth given a storage budget. For Path ORAM we choose $Z = 4$ (increasing Z strictly hurts bandwidth) and tree-top cache to fill remaining space. For Ring ORAM we adjust Z, A and S, tree-top cache and apply the XOR technique.

To simplify the presentation, "client storage" includes all ORAM data structures except for the position map – which has the same space/bandwidth cost for both Path ORAM and Ring ORAM. We remark that applying the recursion technique (Section 3.7) to get a small on-chip position map is cheap for reasonably large blocks. For example, recursing the on-chip position map down to 256 KiloBytes of space when the data block size is 4 KiloBytes increases overall bandwidth for Ring ORAM and Path ORAM by $< 3\%$.

The high order bit is that across different block sizes and client storage budgets, Ring ORAM consistently reduces overall bandwidth relative to Path ORAM by 2-2.7×. We give a summary of these results for several representative client storage budgets in Table 5. We remark that for smaller block sizes, Ring ORAM's improvement over Path ORAM ($\sim 2\times$ for 64 Byte blocks) is smaller relative to when we use larger blocks ($2.7\times$ for 4 KiloByte blocks). The reason is that with small blocks, the cost to read bucket metadata cannot be ignored, forcing Ring ORAM to use smaller Z.

[2] The possibility that a bucket needs to be early reshuffled twice before an eviction is negligible.

		Online, Overall Bandwidth overhead		
Block Size (Bytes)	Z, A (Ring ORAM only)	**Ring ORAM**	**Ring ORAM (XOR)**	Path ORAM
64	$10, 11$	$48\times, 144\times$	$24\times, 118\times$	$120\times, 240\times$
4096	$33, 48$	$20\times, 82\times$	$\sim 1\times, 60\times$	$80\times, 160\times$

Table 5: Breakdown between online and offline bandwidth given a client storage budget of 1000× the block size for several representative points (Section 6.1). Overheads are relative to an insecure system. Parameter meaning is given in Table 2.

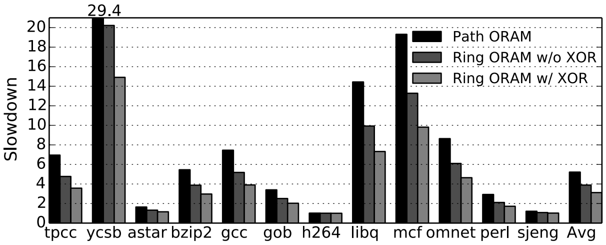

Figure 7: SPEC benchmark slowdown.

6.2 Case Study: Secure Processors

In this study, we show how Ring ORAM improves the performance of secure processors over Path ORAM. We assume the same processor/cache architecture as [5], given in Table 4 of that work. We evaluate a 4 GigaByte ORAM with 64-Byte block size (matching a typical processor's cache line size). Due to the small block size, we parameterize Ring ORAM at $Z = 5, A = 5, X = 2$ to reduce metadata overhead. We use the optimized ORAM recursion techniques [22]: we apply recursion three times with 32-Byte position map block size and get a 256 KB final position map. We evaluate performance for SPEC-int benchmarks and two database benchmarks, and simulate 3 billion instructions for each benchmark. We assume a flat 50-cycle DRAM latency, and compute ORAM latency assuming 128 bits/cycle processor-memory bandwidth. We do not use tree-top caching since it proportionally benefits both Ring ORAM and Path ORAM. Today's DRAM DIMMs cannot perform any computation, but it is not hard to imagine having simple XOR logic either inside memory, or connected to $O(\log N)$ parallel DIMMs so as not to occupy processor-memory bandwidth. Thus, we show results with and without the XOR technique.

Figure 7 shows program slowdown over an insecure DRAM. The high order bit is that using Ring ORAM with XOR results in a geometric average slowdown of 2.8× relative to an insecure system. This is a 1.5× improvement over Path ORAM. If XOR is not available, the slowdown over an insecure system is 3.2×.

We have also repeated the experiment with the unified ORAM recursion technique and its parameters [5]. The geometric average slowdown over an insecure system is 2.4× (2.5× without XOR).

7 Ring ORAM with Large Client Storage

If given a large client storage budget, we can first choose very large A and Z for Ring ORAM, which means bandwidth approaches $2\log N$ (Section 5).[3] Then remaining client storage can be used to tree-top cache (Section 3.6). For example, tree-top caching $t = L/2$ levels requires $O(\sqrt{N})$ storage and bandwidth drops by a factor of 2 to $1 \cdot \log N$—which roughly matches the SSS construction [25].

Burst ORAM [3] extends the SSS construction to handle millions of accesses in a short period, followed by a relatively long idle time where there are few requests. The idea to adapt Ring ORAM to handle bursts is to delay multiple (potentially millions of) EvictPath operations until after the burst of requests. Unfortunately, this strategy means we will experience a much higher early reshuffle rate in levels towards the root. The solution is to coordinate tree-top caching with delayed evictions: For a given tree-top size t, we allow at most 2^t delayed EvictPath operations. This ensures that for levels $\geq t$, the early reshuffle rate matches our analysis in Section 5. We experimentally compared this methodology to the dataset used by Burst ORAM and verified that it gives comparable performance to that work.

8 Related Work

ORAM was first proposed by Goldreich and Ostrovsky [10, 11]. Since then, there have been numerous follow-up works that significantly improved ORAM's efficiency in the past three decades [21, 20, 2, 1, 29, 12, 13, 15, 25, 23, 9, 27, 28]. We have already reviewed two state-of-the-art schemes with different client storage requirements: Path ORAM [27] and the SSS ORAM [25]. Circuit ORAM [28] is another recent tree-based ORAM, which requires only $O(1)$ client storage, but its bandwidth is a constant factor worse than Path ORAM.

Reducing online bandwidth. Two recent works [3, 19] have made efforts to reduce online bandwidth (response time). Unfortunately, the techniques in Burst ORAM [3] do not work with Path ORAM (or more generally any existing tree-based ORAMs). On the

[3] We assume the XOR technique because large client storage implies a file server setting.

other hand, Path-PIR [19], while featuring a tree-based ORAM, employs heavy primitives like Private Information Retrieval (PIR) or even FHE, and thus requires a significant amount of server computation. In comparison, our techniques efficiently achieve $O(1)$ online cost for tree-based ORAMs without resorting to PIR/FHE, and also improve bursty workload performance similar to Burst ORAM.

Subsequent work. Techniques proposed in this paper have been adopted by subsequent works. For example, Tiny ORAM [6] and Onion ORAM [4] used part of our eviction strategy in their design for different purposes.

9 Conclusion

This paper proposes Ring ORAM, the most bandwidth-efficient ORAM scheme for the small (constant or polylog) client storage setting. Ring ORAM is simple, flexible and backed by a tight theoretic analysis.

Ring ORAM is the first tree-based ORAM whose online and overall bandwidth are independent of tree ORAM bucket size. With this and additional properties of the algorithm, we show that Ring ORAM improves online bandwidth by 60× (if simple computation such as XOR is available at memory), and overall bandwidth by 2.3× to 4× relative to Path ORAM. In a secure processor case study, we show that Ring ORAM's bandwidth improvement translates to an overall program performance improvement of 1.5×. By increasing Ring ORAM's client storage, Ring ORAM is competitive in the cloud storage setting as well.

Acknowledgement

This research was partially by NSF grant CNS-1413996 and CNS-1314857, the QCRI-CSAIL partnership, a Sloan Fellowship, and Google Research Awards. Christopher Fletcher was supported by a DoD National Defense Science and Engineering Graduate Fellowship.

References

[1] Boneh, D., Mazieres, D., and Popa, R. A. Remote oblivious storage: Making oblivious RAM practical. Manuscript, `http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf`, 2011.

[2] Damgård, I., Meldgaard, S., and Nielsen, J. B. Perfectly secure oblivious RAM without random oracles. In *TCC* (2011).

[3] Dautrich, J., Stefanov, E., and Shi, E. Burst oram: Minimizing oram response times for bursty access patterns. In *USENIX* (2014).

[4] Devadas, S., van Dijk, M., Fletcher, C. W., Ren, L., Shi, E., and Wichs, D. Onion oram: A constant bandwidth blowup oblivious ram. Cryptology ePrint Archive, 2015. `http://eprint.iacr.org/2015/005`.

[5] Fletcher, C., Ren, L., Kwon, A., van Dijk, M., and Devadas, S. Freecursive oram: [nearly] free recursion and integrity verification for position-based oblivious ram. In *ASPLOS* (2015).

[6] Fletcher, C., Ren, L., Kwon, A., Van Dijk, M., Stefanov, E., Serpanos, D., and Devadas, S. A low-latency, low-area hardware oblivious ram controller. In *FCCM* (2015).

[7] Fletcher, C., Ren, L., Yu, X., Van Dijk, M., Khan, O., and Devadas, S. Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs. In *HPCA* (2014).

[8] Fletcher, C., van Dijk, M., and Devadas, S. Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *STC* (2012).

[9] Gentry, C., Goldman, K. A., Halevi, S., Jutla, C. S., Raykova, M., and Wichs, D. Optimizing oram and using it efficiently for secure computation. In *PET* (2013).

[10] Goldreich, O. Towards a theory of software protection and simulation on oblivious rams. In *STOC* (1987).

[11] Goldreich, O., and Ostrovsky, R. Software protection and simulation on oblivious rams. In *J. ACM* (1996).

[12] Goodrich, M. T., and Mitzenmacher, M. Privacy-preserving access of outsourced data via oblivious ram simulation. In *ICALP* (2011).

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

[14] Islam, M., Kuzu, M., and Kantarcioglu, M. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS* (2012).

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

[16] Liu, C., Harris, A., Maas, M., Hicks, M., Tiwari, M., and Shi, E. Ghostrider: A hardware-software system for memory trace oblivious computation. In *ASPLOS* (2015).

[17] Lorch, J. R., Parno, B., Mickens, J. W., Raykova, M., and Schiffman, J. Shroud: Ensuring private access to large-scale data in the data center. In *FAST* (2013).

[18] Maas, M., Love, E., Stefanov, E., Tiwari, M., Shi, E., Asanovic, K., Kubiatowicz, J., and Song, D. Phantom: Practical oblivious computation in a secure processor. In *CCS* (2013).

[19] Mayberry, T., Blass, E.-O., and Chan, A. H. Efficient private file retrieval by combining oram and pir. In *NDSS* (2014).

[20] OSTROVSKY, R. Efficient computation on oblivious rams. In *STOC* (1990).

[21] OSTROVSKY, R., AND SHOUP, V. Private information storage (extended abstract). In *STOC* (1997).

[22] REN, L., YU, X., FLETCHER, C., VAN DIJK, M., AND DEVADAS, S. Design space exploration and optimization of path oblivious ram in secure processors. In *ISCA* (2013).

[23] SHI, E., CHAN, T.-H. H., STEFANOV, E., AND LI, M. Oblivious ram with $o((\log n)^3)$ worst-case cost. In *Asiacrypt* (2011).

[24] STEFANOV, E., AND SHI, E. Oblivistore: High performance oblivious cloud storage. In *S&P* (2013).

[25] STEFANOV, E., SHI, E., AND SONG, D. Towards practical oblivious RAM. In *NDSS* (2012).

[26] STEFANOV, E., VAN DIJK, M., SHI, E., CHAN, T.-H. H., FLETCHER, C., REN, L., YU, X., AND DEVADAS, S. Path oram: An extremely simple oblivious ram protocol. Cryptology ePrint Archive, 2013. `http://eprint.iacr.org/2013/280`.

[27] STEFANOV, E., VAN DIJK, M., SHI, E., FLETCHER, C., REN, L., YU, X., AND DEVADAS, S. Path oram: An extremely simple oblivious ram protocol. In *CCS* (2013).

[28] WANG, X. S., CHAN, T.-H. H., AND SHI, E. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. Cryptology ePrint Archive, 2014. `http://eprint.iacr.org/2014/672`.

[29] WILLIAMS, P., AND SION, R. Single round access privacy on outsourced storage. In *CCS* (2012).

[30] WILLIAMS, P., SION, R., AND TOMESCU, A. Privatefs: A parallel oblivious file system. In *CCS* (2012).

[31] YU, X., FLETCHER, C. W., REN, L., VAN DIJK, M., AND DEVADAS, S. Generalized external interaction with tamper-resistant hardware with bounded information leakage. In *CCSW* (2013).

[32] ZHUANG, X., ZHANG, T., AND PANDE, S. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. In *ASPLOS* (2004).

A Bucket Structure

Table 6 lists all the fields in a Ring ORAM bucket and their size. We would like to make two remarks. First, only the data fields are permuted and that permutation is stored in ptrs. Other bucket fields do not need to be permuted because when they are needed, they will be read in their entirety. Second, count and valids are stored in plaintext. There is no need to encrypt them since the server can see which bucket is accessed (deducing count for each bucket), and which slot is accessed in each bucket (deducing valids for each bucket). In fact, if the server can do computation and is trusted to follow the protocol faithfully, the client can let the server update count and valids. All the other structures should be probabilistically encrypted.

Algorithm 5 Helper functions.

```
count, valids, addrs, leaves, ptrs, data are fields of the
input bucket in each of the following three functions

function GetBlockOffset(bucket, $a$)
    read in valids, addrs, ptrs
    decrypt addrs, ptrs
    for $j \leftarrow 0$ to $Z-1$ do
        if $a =$ addrs[$j$] and valids[ptrs[$j$]] then
            return ptrs[$j$]                ▷ block of interest
    return a pointer to a random valid dummy

function ReadBucket(bucket)
    read in valids, addrs, leaves, ptrs
    decrypt addrs, leaves, ptrs
    $z \leftarrow 0$            ▷ track # of remaining real blocks
    for $j \leftarrow 0$ to $Z-1$ do
        if valids[ptrs[$j$]] then
            data′ ← read and decrypt data[ptrs[$j$]]
            $z \leftarrow z+1$
            if addrs[$j$] $\neq \perp$ then
                block ← (addr[$j$], leaf[$j$], data′)
                Stash ← Stash ∪ block
    for $j \leftarrow z$ to $Z-1$ do
        read a random valid dummy

function WriteBucket(bucket, Stash)
    find up to $Z$ blocks from Stash that can reside
    in this bucket, to form addrs, leaves, data′
    ptrs ← PRP($0, Z+S$)              ▷ or truly random
    for $j \leftarrow 0$ to $Z-1$ do
        data[ptrs[$j$]] ← data′[$j$]
    valids ← $\{1\}^{Z+S}$
    count ← 0
    encrypt addrs, leaves, ptrs, data
    write out count, valids, addrs, leaves, ptrs, data
```

Having defined the bucket structure, we can be more specific about some of the operations in earlier sections. For example, in Algorithm 2 Line 5 means reading $\mathcal{P}(l,i)$.data[offset], and Line 6 means setting $\mathcal{P}(l,i)$.valids[offset] to 0.

Now we describe the helper functions in detail. GetBlockOffset reads in the valids, addrs, ptrs field, and looks for the block of interest. If it finds the block of interest, meaning that the address of a still valid block matches the block of interest, it returns the permuted location of that block (stored in ptrs). If it does not find the block of interest, it returns the permuted location of a random valid dummy block.

Notation	Size (bits)	Meaning
count	$\log(S)$	# of times this bucket has been touched by ReadPath since it was last shuffled
valids	$(Z+S)*1$	Indicates whether each of the $Z+S$ blocks is valid
addrs	$Z*\log(N)$	Address for each of the Z (potentially) real blocks
leaves	$Z*L$	Leaf label for each of the Z (potentially) real blocks
ptrs	$Z*\log(Z+S)$	Offset in the bucket for each of the Z (potentially) real blocks
data	$(Z+S)*B$	Data field for each of the $Z+S$ blocks, **permuted** according to ptrs
EncSeed	λ (security parameter)	Encryption seed for the bucket; count and valids are stored in the clear

Table 6: Ring ORAM bucket format. All logs are taken to their ceiling.

ReadBucket reads all of the remaining real blocks in a bucket into the stash. For security reasons, ReadBucket always reads *exactly* Z blocks from that bucket. If the bucket contains less than Z valid real blocks, the remaining blocks read out are random valid dummy blocks. Importantly, since we allow at most S reads to each bucket before reshuffling it, it is guaranteed that there are at least Z valid (real + dummy) blocks left that have not been touched since the last reshuffle.

WriteBucket evicts as many blocks as possible (up to Z) from the stash to a certain bucket. If there are $z' \leq Z$ real blocks to be evicted to that bucket, $Z+S-z'$ dummy blocks are added. The $Z+S$ blocks are then randomly shuffled based on either a truly random permutation or a Pseudo Random Permutation (PRP). The permutation is stored in the bucket field ptrs. Then, the function resets count to 0 and all valid bits to 1, since this bucket has just been reshuffled and no blocks have been touched. Finally, the permuted data field along with its metadata are encrypted (except count and valids) and written out to the bucket.

B Proof of the Lemmas

To prove Lemma 1, we made a little change to the Ring ORAM algorithm. In Ring ORAM, a ReadPath operation adds the block of interest to the stash and replaces it with a dummy block in the tree. Instead of making the block of interest in the tree dummy, we turn it into a *stale* block. On an EvictPath operation to path l, all the stale blocks that are mapped to leaf l are turned into dummy blocks. Stale blocks are treated as real blocks in both $\mathsf{ORAM}_L^{Z,A}$ and $\mathsf{ORAM}_L^{\infty,A}$ (including G_Z) until they are turned into dummy blocks. Note that this trick of stale blocks is only to make the proof go through. It hurts the stash occupancy and we will not use it in practice. With the stale block trick, we can use induction to prove Lemma 1.

Proof of Lemma 1. Initially, the lemma obviously holds. Suppose $G_{\mathcal{S}_Z}(\mathcal{S}_\infty) = \mathcal{S}_Z$ after some accesses. We need to show that $G_{\mathcal{S}'_Z}(\mathcal{S}_\infty) = \mathcal{S}'_Z$ where $\mathcal{S}'_Z$ and $\mathcal{S}'_\infty$ are the states after the next operation (either ReadPath or EvictPath). A ReadPath operation adds a block to the stash (the root bucket) for both $\mathsf{ORAM}_L^{Z,A}$ and $\mathsf{ORAM}_L^{\infty,A}$, and does not move any blocks in the tree except turning a real block into a stale block. Since stale blocks are treated as real blocks, $G_{\mathcal{S}'_Z}(\mathcal{S}_\infty) = \mathcal{S}'_Z$ holds.

Now we show the induction holds for an EvictPath operation. Let EP_l^Z be an EvictPath operation to $\mathcal{P}(l)$ (path l) in $\mathsf{ORAM}_L^{Z,A}$ and EP_l^∞ be an EvictPath operation to $\mathcal{P}(l)$ in $\mathsf{ORAM}_L^{\infty,A}$. Then, $\mathcal{S}'_Z = \mathsf{EP}_l^Z(\mathcal{S}_Z)$ and $\mathcal{S}'_\infty = \mathsf{EP}_l^\infty(\mathcal{S}_\infty)$. Note that EP_l^Z has the same effect as EP_l^∞ followed by post-processing, so

$$\begin{aligned}\mathcal{S}'_Z = \mathsf{EP}_l^Z(\mathcal{S}_Z) &= G_{\mathcal{S}'_Z}(\mathsf{EP}_l^\infty(\mathcal{S}_Z)) \\ &= G_{\mathcal{S}'_Z}(\mathsf{EP}_l^\infty(G_{\mathcal{S}_Z}(\mathcal{S}_\infty)))\end{aligned}$$

The last equation is due to the induction hypothesis.

It remains to show that

$$G_{\mathcal{S}'_Z}(\mathsf{EP}_l^\infty(G_{\mathcal{S}_Z}(\mathcal{S}_\infty))) = G_{\mathcal{S}'_Z}(\mathsf{EP}_l^\infty(\mathcal{S}_\infty)),$$

which is $G_{\mathcal{S}'_Z}(\mathcal{S}'_\infty)$. To show this, we decompose G into steps for each bucket, i.e., $G_{\mathcal{S}_Z}(\mathcal{S}_\infty) = g_1 g_2 \cdots g_{2^{L+1}}(\mathcal{S}_\infty)$ where g_i processes bucket b_i^∞ in reference to b_i^Z. Similarly, we decompose $G_{\mathcal{S}'_Z}$ into $g'_1 g'_2 \cdots g'_{2^{L+1}}$ where each g'_i processes bucket $b_i'^\infty$ of $\mathcal{S}'_\infty$ in reference to $b_i'^Z$ of $\mathcal{S}'_Z$. We now only need to show that for any $0 < i < 2^{L+1}$, $G_{\mathcal{S}'_Z}(\mathsf{EP}_l^\infty(g_1 g_2 \cdots g_i(\mathcal{S}_\infty))) = G_{\mathcal{S}'_Z}(\mathsf{EP}_l^\infty(g_1 g_2 \cdots g_{i-1}(\mathcal{S}_\infty)))$. This is obvious if we consider the following three cases separately:

1. If $b_i \in \mathcal{P}(l)$, then g_i before EP_l^∞ has no effect since EP_l^∞ moves all blocks on $\mathcal{P}(l)$ into the stash before evicting them to $\mathcal{P}(l)$.
2. If $b_i \notin \mathcal{P}(l)$ and $b_{i/2} \notin \mathcal{P}(l)$ (neither b_i nor its parent is on Path l), then g_i and EP_l^∞ touch non-overlapping buckets and do not interfere with each other. Hence, their order can be swapped, $G_{\mathcal{S}'_Z}(\mathsf{EP}_l^\infty(g_0 g_1 g_2 \cdots g_i(\mathcal{S}_\infty))) = G_{\mathcal{S}'_Z} g_i(\mathsf{EP}_l^\infty(g_0 g_1 g_2 \cdots g_{i-1}(\mathcal{S}_\infty)))$. Furthermore,

$b_i^Z = b_i'^Z$ (since EP_l^∞ does not change the content of b_i), so g_i has the same effect as g_i' and can be merged into $G_{\mathcal{S}_Z'}$.

3. If $b_i \notin \mathcal{P}(l)$ but $b_{i/2} \in \mathcal{P}(l)$, the blocks moved into $b_{i/2}$ by g_i will stay in $b_{i/2}$ after EP_l^∞ since $b_{i/2}$ is the highest intersection (towards the leaf) that these blocks can go to. So g_i can be swapped with EP_l^∞ and can be merged into $G_{\mathcal{S}_Z'}$ as in the second case.

We remind the readers that because we only remove stale blocks that are mapped to $\mathcal{P}(l)$, the first case is the only case where some stale blocks in b_i may turn into dummy blocks. And the same set of stale blocks are removed from $\mathsf{ORAM}_L^{Z,A}$ and $\mathsf{ORAM}_L^{\infty,A}$.

This shows

$$\begin{aligned} G_{\mathcal{S}_Z'}\left(\mathsf{EP}_l^\infty\left(G_{\mathcal{S}_Z}\left(\mathcal{S}_\infty\right)\right)\right) &= G_{\mathcal{S}_Z'}\left(\mathsf{EP}_l^\infty\left(\mathcal{S}_\infty\right)\right) \\ &= G_{\mathcal{S}_Z'}\left(\mathcal{S}_\infty'\right) \end{aligned}$$

and completes the proof. □

The proof of Lemma 2 remains unchanged from the Path ORAM paper [27], and is replicated here for completeness.

Proof of Lemma 2. If part: Suppose $T \in \mathsf{ORAM}_L^{\infty,A}$ and $X(T) > c(T) + R$. Observe that G can assign the blocks in a bucket only to an ancestor bucket. Since T can store at most $c(T)$ blocks, more than R blocks must be assigned to the stash by G.

Only if part: Suppose that $\mathsf{st}\left(G_{\mathcal{S}_Z}\left(\mathcal{S}_\infty\right)\right) > R$. Let T be the maximal rooted subtree such that all the buckets in T contain exactly Z blocks after post-processing G. Suppose b is a bucket not in T. By the maximality of T, there is an ancestor (not necessarily proper ancestor) bucket b' of b that contains less than Z blocks after post-processing, which implies that no block from b can go to the stash. Hence, all blocks that are in the stash must have originated from T. Therefore, it follows that $X(T) > c(T) + R$. □

Proof of Lemma 3. For a bucket b in $\mathsf{ORAM}_L^{\infty,A}$, define $Y(b)$ to be the number of blocks in b before post-processing. It suffices to prove that $\forall b \in \mathsf{ORAM}_L^{\infty,A}$, $E[Y(b)] \leq A/2$.

If b is a leaf bucket, the blocks in it are put there by the last EvictPath operation to that leaf/path. Note that only real blocks could be put in b by that operation, although some of them may have turned into stale blocks. Stale blocks can never be moved into a leaf by an EvictPath operation, because that EvictPath operation would remove all the stale blocks mapped to that leaf. There are at most N distinct real blocks and each block has a probability of 2^{-L} to be mapped to b independently. Thus $E[Y(b)] \leq N \cdot 2^{-L} \leq A/2$.

If b is not a leaf bucket, we define two variables m_1 and m_2: the last EvictPath operation to b's left child is the m_1-th EvictPath operation, and the last EvictPath operation to b's right child is the m_2-th EvictPath operation. Without loss of generality, assume $m_1 < m_2$. We then time-stamp the blocks as follows. When a block is accessed and remapped, it gets time stamp m^*, which is the number of EvictPath operations that have happened. Blocks with $m^* \leq m_1$ will not be in b as they will go to either the left child or the right child of b. Blocks with $m^* > m_2$ will not be in b as the last access to b (m_2-th) has already passed. Therefore, only blocks with time stamp $m_1 < m^* \leq m_2$ will be put in b by the m_2-th access. (Some of them may be accessed again after the m_2-th access and become stale, but this does not affect the total number of blocks in b as stale blocks are treated as real blocks.) There are at most $d = A|m_1 - m_2|$ such blocks, and each goes to b independently with a probability of $2^{-(i+1)}$, where i is the level of b. The deterministic nature of evictions in Ring ORAM ensures $|m_1 - m_2| = 2^i$. (One way to see this is that a bucket b at level i will be written every 2^i EvictPath operations, and two consecutive EvictPath operations to b always travel down the two different children of b.) Therefore, $E[Y(b)] \leq d \cdot 2^{-(i+1)} = A/2$ for any non-leaf bucket as well. □