Ring ORAM: Closing the Gap Between Small and Large Client Storage Oblivious RAM

Ling Ren†, Christopher W. Fletcher†, Albert Kwon†, Emil Stefanov°, Elaine Shi * Marten van Dijk‡, Srinivas Devadas†

† Massachusetts Institute of Technology – {renling, cwfletch, kwonal, devadas}@mit.edu
° University of California, Berkeley – emil@berkeley.edu
* University of Maryland, College Park – elaine@cs.umd.edu
‡ University of Connecticut – vandijk@engr.uconn.edu

[1]

***Abstract*—We present Ring ORAM, a simple and low-latency ORAM construction that can be parameterized for either small or large client storage. Simply by tuning parameters, Ring ORAM matches or exceeds the performance of the best-known small and large client storage schemes and can achieve a constant factor online bandwidth overhead over insecure systems.**

We evaluate Ring ORAM in theory and in practice. On the theory side, we prove that Ring ORAM matches the asymptotic bandwidth and client storage of Path ORAM, the prior-art scheme for small client storage. Tuning parameters for small client storage, Ring ORAM reduces overall bandwidth relative to Path ORAM by a factor of $2.7\times$ and reduces online bandwidth to constant (a $57\times$ improvement over Path ORAM given realistic parameters). Tuning parameters for large client storage, Ring ORAM outperforms Path ORAM (which is given equal storage) by $4.5\times$ and SSS ORAM, the prior-art scheme for large client storage, by 16-33%.

Using secure processors as a case study for small client storage, Ring ORAM on average reduces ORAM response time by nearly $5\times$ and improves workload completion time by $2.75\times$, relative to Path ORAM. In the large storage setting, running an enterprise file system trace with bursty requests, Ring ORAM adds negligible overhead to response time given a > 100 Mbps network bandwidth. By comparison, Burst ORAM incurs large overheads in response time unless network bandwidth > 350 Mbps. These results suggest that Ring ORAM is a compelling construction in both large client storage (e.g., file server) and small client storage (e.g., remote secure processor) settings.

I. Introduction

With cloud computing and storage becoming increasingly popular, privacy of users' sensitive data has become a large concern in computation and storage outsourcing. In an ideal setting, users would like to "throw their encrypted data over the wall" to a cloud service without revealing anything from within that data to the server. It is well known, however, that encryption is not enough to get privacy. The user's access pattern has been shown to reveal a lot of information about the encrypted files [11] or the private user data being computed upon [28], [14].

[1]A preliminary version of this paper appears in Cryptology ePrint Archive Report 2014/431.

Oblivious RAM (ORAM) is a cryptographic primitive that *completely* eliminates the information leakage in memory access trace. In ORAM schemes, a client stores its data in encrypted and shuffled form on an untrusted server. On each access, the client reads the untrusted memory, reshuffles that memory and possibly updates some state in trusted local storage called *the client storage*. Under ORAM, any memory access pattern is computationally indistinguishable from any other access pattern of the same length.

The most efficient ORAM scheme when a large amount of client storage is available is the SSS construction [22] and the most efficient scheme for small client storage is Path ORAM [24]. In this paper, we define small client storage to be poly-logarithmic.

In practice, large vs. small client storage leads to very different use cases. For example, in the oblivious file server setting [26], [21], [4], clients usually have ample local storage and use the SSS construction for its better bandwidth. On the other hand, as on-chip storage is scarce, secure processor proposals [5], [14] have adopted Path ORAM due to its simplicity and small client storage. Therefore, Path ORAM and SSS ORAM schemes are considered 'best in class' for their respective settings only.

A. Our Contributions

This paper proposes Ring ORAM, the most bandwidth-efficient ORAM protocol to date for both small (poly-log) *and* large client storage. Ring ORAM has the following intriguing properties, which we quantify in Table I:

1) **Multi-paradigm Oblivious RAM.** Ring ORAM combines the best qualities from the SSS ORAM and Path ORAM. Most of the bandwidth/latency improving optimizations used by the SSS ORAM (and subsequent work Burst ORAM [4]) are immediately applicable to Ring ORAM. At the same time, Ring ORAM is fundamentally a tree-based ORAM [20] and therefore inherits the simplicity and small client storage that are typical for tree-based ORAMs.
2) **Efficiency and flexible parameters.** Partly due to the above property, Ring ORAM offers appealing perfor-

TABLE I: **Our contributions.** Overheads are relative to an insecure system. Online bandwidth is the bandwidth needed to serve a request and overall bandwidth includes background work. XOR refers to the XOR technique from [4] and level comp refers to level compression from [22]. We show Ring ORAM without the XOR technique for small client storage in case server computation is not available in that setting. The constant c in the large client storage case is very small for realistic block sizes: i.e., for this parameterization the cN term constitutes $1/16$ of total client storage. Bandwidth in the small client storage case does not include the cost of recursion [20] which is negligible in practice for large (e.g. 4 KByte) block sizes.

	Practical (overheads are relative to insecure), Asymptotic						
	Online Bandwidth		Overall Bandwidth		Client Storage		Server Storage
Large client storage (ORAM capacity = 64 TeraBytes, Block size 256 KBytes)							
SSS ORAM [22] (level comp)	$7.8\times$	$O(\log N)$	$31.2\times$	$O(\log N)$	16 GBytes	$O(\sqrt{N})+cN$	$3.2N$
SSS ORAM [22] (XOR)	$1\times$	$O(1)$	$35.7\times$	$O(\log N)$	16 GBytes	$O(\sqrt{N})+cN$	$3.2N$
Path ORAM [24]	$60\times$	$O(\log N)$	$120\times$	$O(\log N)$	16 GBytes	$O(\sqrt{N})+cN$	$8N$
Ring ORAM (**this paper**, XOR)	$1\times$	$O(1)$	$26.8\times$	$O(\log N)$	16 GBytes	$O(\sqrt{N})+cN$	$6N$
Small client storage (ORAM capacity = 1 TeraByte, Block size 4 KBytes)							
Path ORAM	$80\times$	$O(\log N)$	$160\times$	$O(\log N)$	3.1 MBytes	$O(\log N)\omega(1)$	$8N$
Ring ORAM (**this paper**)	$20.4\times$	$O(\log N)$	$79.3\times$	$O(\log N)$	3.1 MBytes	$O(\log N)\omega(1)$	$6N$
Ring ORAM (**this paper**, XOR)	$1.4\times$	$O(1)$	$59.7\times$	$O(\log N)$	3.1 MBytes	$O(\log N)\omega(1)$	$6N$

mance at both settings. Given small client storage, Ring ORAM improves the overall bandwidth of Path ORAM by $2-2.7\times$ and online bandwidth (i.e., bandwidth to serve a request) by $>50\times$, for a practical parameter setting. When given large client storage, Ring ORAM improves over Path ORAM (which is given equal storage) by $4.5\times$ and the SSS ORAM by 16-33% (depending on parameters).

3) **Simplified theoretical analysis.** Similar to Path ORAM, Ring ORAM achieves negligible failure probability with a $O(\log N)\omega(1)$ size stash. Our novel proof technique based on properties of the Ring ORAM eviction algorithm leads to a much simpler stash analysis than that of Path ORAM. We also note that the proof of Lemma 1 in [24], a crucial lemma for both Path ORAM and this paper, is incomplete, though the lemma itself is correct. We give a rigorous proof for that lemma in this paper.

We evaluate Ring ORAM in two case studies to show its performance. First, in an enterprise file server (large client storage setting) with bursty requests, given medium network bandwidth (>100 Mbps), Ring ORAM's response time is approximately the same as a baseline system and significantly outperforms Burst ORAM [4] (unless Burst ORAM is given >350 Mbps bandwidth). Second, in a secure processor (small client storage setting) running SPEC and database workloads, Ring ORAM improves response time by nearly $5\times$, which translates to a $2.75\times$ speedup in program completion time compared to Path ORAM. These results show how Ring ORAM is a compelling scheme in both large (file server) and small (secure processor) client storage settings.

B. Paper Organization

In § II, we give formal definitions for ORAM, related work and a background for the small and large client storage settings. § III explains the Ring ORAM protocol in detail. § IV describes scheduling optimizations for Ring ORAM. § V gives a complete formal analysis for bounding Ring ORAM's client storage. § VI gives a methodology for setting Ring ORAM parameters. In § VII, we compare Ring ORAM to prior work in terms of bandwidth vs. client storage and two realistic case studies. Finally, we conclude in § VIII.

II. Background and Related Work

A. An Overview of ORAM Schemes

ORAM schemes consider a trusted client that wishes to store private data on an untrusted server. The goal of ORAM is complete access pattern obfuscation: the server should learn no information about the data read/written, the sequence of addresses or operations (reads/writes). For this paper we follow the standard security definition for ORAM.

Definition 1. (ORAM Definition) *Let*

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

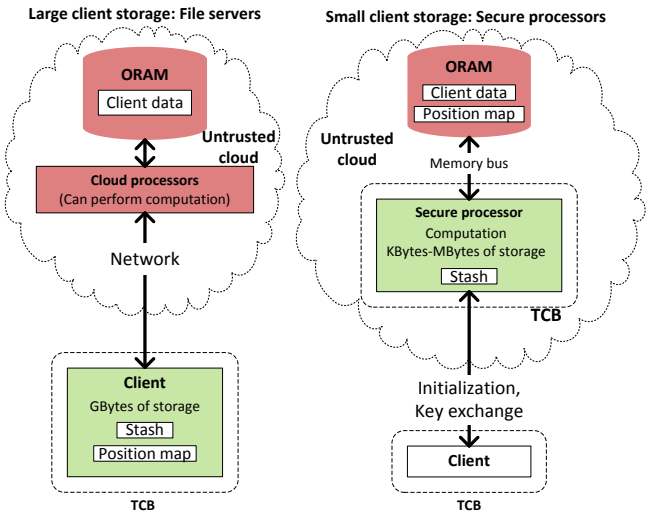
denote a data sequence of length M, where op_i denotes whether the i-th operation is a read or write, addr_i denotes the address for that access and data_i denotes the data (if a write). Let $\mathsf{ORAM}(\overleftarrow{y})$ be the resulting randomized data request sequence of an ORAM algorithm. The 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 consistant with $\overleftarrow{y}$ (i.e., the ORAM behaves like a valid RAM) with overwhelming probability.

ORAM was first proposed by Goldreich and Ostrovsky [7], [8]. Since then, there have been numerous follow-up works that significantly improved ORAM's efficiency in the past three decades [17], [16], [3], [1], [25], [9], [10], [12], [22], [20], [6], [24].

To compare ORAM schemes, we consider their *online bandwidth*, *offline bandwidth* and *client storage*. Online bandwidth is the work required to serve a client request while offline bandwidth can be performed in the background. Thus, online bandwidth is more important in some settings (e.g., for a workload with bursts of requests and sufficient idle time to hide background work). When we say overall bandwidth we mean the sum of online and offline bandwidth. Client storage is the amount of trusted local memory required at the client side to manage the ORAM protocol.

Under these metrics, the two state-of-the-art schemes in practical settings are the SSS ORAM [22] and Path ORAM [24]. Both schemes achieve $O(\log N)$ overall bandwidth, the asymptotic lower bound found by Goldreich and Ostrovsky [7], [8]. The SSS ORAM has a constant factor of around 1 in overall bandwidth, but requires at least $O(\sqrt{N})$ client storage. Burst ORAM extends the SSS construction [22] by reducing its online bandwidth to $O(1)$. Path ORAM has a larger constant factor in its overall bandwidth—at least 8, roughly half of which is online bandwidth—but only needs poly-logarithmic client storage. Since Path ORAM, several schemes [15] have been developed for constant client storage but make additional assumptions such as such as PIR/FHE at the server. Thus, we do not compare to them in this paper.

The bandwidth and storage characteristics of Path ORAM and SSS ORAM make each the 'scheme of choice' in distinct settings called *small client storage* and *large client storage*. The proposal of this paper, Ring ORAM, matches or exceeds each scheme in its preferred setting. For context, we now discuss some characteristics and concrete examples for both settings (summarized in Figure 1).



(a) File server setting (b) Secure processor setting

Fig. 1: ORAM usage settings. The position map and stash are ORAM client storage data structures that will be explained in § III.

B. Large Client Storage (e.g., Oblivious File Servers)

An example of the large client storage setting is remote oblivious file servers (Figure 1a). Previous works in this area include [26], [13], [21], [4]. In this setting, the client runs on a local machine and interacts with a remote storage provider (e.g., a datacenter) to retrieve data obliviously. The trusted computing base (TCB) is the client machine, and the client can use its main memory or disk for client storage, typically tens of GBytes (still small compared to a tens of TeraBytes ORAM). Given this large storage, the preferred ORAM scheme in this setting is the SSS construction [22], which was implemented in Oblivistore [21] and extended to minimize online bandwidth in Burst ORAM [4].

TABLE II: ORAM parameters and notations.

Notation	Meaning
N	Number of real data blocks in tree
L	Depth of Path 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 $\mathcal{P}(l,i)$

Since ORAM accesses typically are reading/writing files, the ORAM block size is typically large (at least 4 KBytes). Since data is sent over a network with potentially large latency, it is desirable to minimize the number of round-trip operations. Further, the remote server is usually capable of performing some public computation on the client's data to save network bandwidth or round-trips.

C. Small Client Storage (e.g., Secure Processors)

An example small client setting is when the client is a remote secure processor (Figure 1b). Client storage is restricted to a processor's on-chip storage which is on the order of hundreds of KBytes to MBytes. The ORAM client logic is typically implemented directly in hardware and needs to be simple. Therefore, secure processors [14], [5], [27], [19] have adopted Path ORAM [24] for its small client storage and simplicity.

Since secure processor main memory is close to the processor, round trip latency is typically small (tens of nanoseconds). At the same time, main memory (e.g., stock DRAM) typically has no computation ability beyond simply writing and reading data, which may prohibit some optimizations used in the file server setting.

III. Ring ORAM Protocol

A. 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 II.

Server storage is organized as a binary tree of buckets where each bucket is made up a small number of slots which 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 leaf is given a unique indentifier. Each bucket has $Z + S$ slots and a small amount of header information that we will detail in § III-F. Of these slots, up to Z slots may contain real blocks and the remaining S slots are reserved for dummy blocks.

Server storage size. Our theoretical analysis in § V will show that to store N blocks in Ring ORAM, the physical ORAM tree needs roughly $6N$ to $8N$ slots.

Client storage is made up 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. The stash buffers blocks that have not been evicted to the ORAM tree and additionally stores $Z(L+1)$ blocks for shuffling operations.

Client storage size. We will prove in § V that stash size is bounded by $O(\log N)\omega(1)$ with high probability. The position map stores $N * L$ bits, but can be squashed to constant storage using the standard recursion trick (§ III-I).

Main invariants. Ring ORAM has two main invariants:

1) (Same as Path ORAM): Every block tracked by the position map is mapped to a leaf bucket chosen uniformly at random in the ORAM tree. If a block a is mapped to leaf l by the position map, 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) For every bucket in the tree, the physical positions of the $Z + S$ dummy and real blocks in each bucket are permuted randomly 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.

B. Access and Eviction Operations

Algorithm 1 Non-recursive Ring ORAM.

```
1:  function ACCESS(a, op, data')
2:      Global/persistent variables: round
3:      l' ← UniformRandom(0, 2^L − 1)
4:      l ← PositionMap[a]
5:      PositionMap[a] ← l'
6:      data ← ReadPath(l, a)
7:      if data = ⊥ then
8:          ▷ If block a is not found on path l, it must
9:          be in Stash ◁
10:         data ← read and remove a from Stash
11:     if op = read then
12:         return data to client
13:     if op = write then
14:         data ← data'
15:     Stash ← Stash ∪ (a, l', data)
16:     round ← round + 1 mod A
17:     if round =? 0 then
18:         EvictPath()
19:     EarlyReshuffle(l)
```

The Ring ORAM access protocol is shown in Algorithm 1. At a very high level, this algorithm is similar to the SSS ORAM [22], although the mechanics of each access and eviction are very different. In particular, each access is four steps:

Position Map lookup (Lines 3-5): Lookup 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 step is unchanged from previous tree-based ORAMs [20], [24].

We denote path l as $\mathcal{P}(l)$, and the i-th bucket (towards the root) on that path as $\mathcal{P}(l, i)$. Note that a bucket can have multiple aliases (e.g., $\mathcal{P}(l, 0)$ is the root bucket for any l).

Read Path (Lines 6-15): The $\mathsf{ReadPath}(l, a)$ 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. Our ReadPath operation differs significantly from Path ORAM in that we read out only one block—from a *random* location—from each bucket. This lowers the bandwidth overhead of ReadPath to $L + 1$ blocks (the number of buckets on a path) or even a single block if further optimizations are applied (§ III-H).

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 far down towards the tree leaves as possible. The sole purpose of an eviction operation is to push blocks back to the binary tree from the stash.

Unlike Path ORAM, EvictPath for Ring ORAM 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 less eviction operations than Path ORAM (which performs one eviction operation after each access).

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. The helper functions needed in these three subroutines are given in Algorithm 5 and explained in § III-F after we describe the structure of buckets. We also explain the security for each subroutine in § III-G.

C. Read Path Operation

Algorithm 2 ReadPath procedure. Helper functions are defined in Algorithm 5.

```
1:  function ReadPath(l, a)
2:      data ← ⊥
3:      for i ← 0 to L do
4:          offset ← GetBlockOffset(P(l, i), a)
5:          data' ← P(l, i, offset)
6:          Invalidate P(l, i, offset)
7:          if data' ≠ ⊥ then
8:              data ← data'
9:          P(l, i).count ← 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.

Because the position map only tracks the *path* containing the block of interest (§ III-A) the client does not know where in each bucket to look for the block of interest. Thus, for each bucket we must read some metadata into client memory that maps each real block in the bucket to one of the $Z + S$ slots (Lines 4). The details of GetBlockOffset will be presented in § III-F after we describe the bucket format. The metadata is small compared to reasonably large block sizes. When the block size is small, however, reading metadata becomes a limiting factor in bandwidth (§ VII-A2). Once we know the offset into the bucket, Line 5 reads the block in the slot, and invalidates it.

Each bucket also has a counter which tracks how many times it is accessed (Line 9). If a bucket is read too many ($> S$) times, it may run out of dummy blocks (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[2]. Therefore, we need to reshuffle single buckets on-demand as soon as they are touched more than S times. We call this step EarlyReshuffle and describe it in § III-E.

D. Evict Path Operation

Algorithm 3 EvictPath procedure. Helper functions are defined in Algorithm 5.

```
1: function EvictPath
2:     Global/persistant variables G initialized to 0
3:     l ← G mod 2^L
4:     G ← G + 1
5:     for i ← 0 to L do
6:         Stash ← Stash ∪ ReadBucket(P(l, i))
7:     for i ← L to 0 do
8:         WriteBucket(P(l, i), Stash)
9:         P(l, i).count ← 0
```

The EvictPath routine is shown in Algorithm 3. As mentioned, evictions are scheduled statically: one eviction 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 9) reflects this: we wish to fill the deepest buckets as full as possible. Thus, EvictPath is like a Path ORAM access where no block is accessed and therefore no block is remapped to a new leaf.

[2]We could consider reading (and invalidating) a valid real block from that bucket, similar to early cache-ins in Oblivistore [21], but this adds extra blocks into the stash, and will lead to inferior parameter settings.

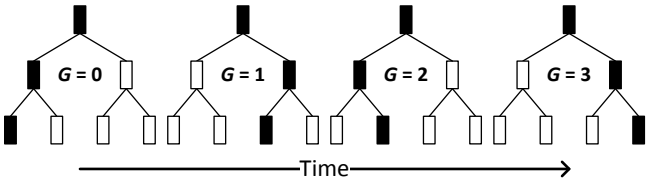


Fig. 2: Reverse-lexicographic order of paths used by EvictPath. After the $G = 3$ path is evicted to, the order repeats.

However, there are two important differences from Path ORAM's eviction operation. First, evictions in Ring ORAM are performed to paths in a specific order called the **reverse-lexicographic order** shown in Figure 2, and first proposed by Gentry et al. [6]. 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 likely to be less efficient. 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 § V.

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

E. Early Reshuffle Operation

Algorithm 4 EarlyReshuffle procedure. Helper functions are defined in Algorithm 5.

```
1: function EarlyReshuffle(l)
2:     for i ← 0 to L do
3:         if P(l, i).count ≥ S then
4:             Stash ← Stash ∪ ReadBucket(P(l, i))
5:             WriteBucket(P(l, i), Stash)
6:             P(l, i).count ← 0
```

In some sense EvictPath are "statically scheduled" reshuffles. But due on randomness, a bucket can be touched $> S$ times by ReadPath operations before it is processed by the scheduled EvictPath. In this case, we call EarlyReshuffle on that bucket to reshuffle it. After each ORAM access, EarlyReshuffle goes over all the buckets on the accessed path, and reshuffles all the buckets that have been accessed more than S times (see § III-C) by performing ReadBucket and WriteBucket. Again, these two helper functions will be explained in § III-F. We note that though S does not affect the security (§ III-G), it has an impact on the performance (how often we shuffle). We discuss how to select S in § VI-A.

F. Bucket Structure

Table III lists all the fields in a Ring ORAM bucket and their size. We would like to make two remarks. First, only the

TABLE III: Ring ORAM bucket format. All logs are taken to their ceiling.

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

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, as mentioned in § III-E), and which slot is accessed in a 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.

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 (stored in ptrs) of that block. If it does not find the block of interest, it returns the permuted location of a random valid dummy block.

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 which 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, an additional $Z+S-z'$ dummy blocks are added. The $Z+S$ blocks are then randomly shuffled based on 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.

Algorithm 5 Helper functions.

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

```
1:  function GetBlockOffset(bucket, a)
2:      read in valids, addrs, ptrs
3:      decrypt addrs, ptrs
4:      for j ← 0 to Z − 1 do
5:          if a = addrs[j] and valids[ptrs[j]] then
6:              return ptrs[j]                    ▷ block of interest
        return a pointer to a random valid dummy

1:  function ReadBucket(bucket)
2:      read in valids, addrs, leaves, ptrs
3:      decrypt addrs, leaves, ptrs
4:      z ← 0                          ▷ track # of remaining real blocks
5:      for j ← 0 to Z − 1 do
6:          if valids[ptrs[j]] then
7:              data′ ← read and decrypt data[ptrs[j]]
8:              z ← z + 1
9:              if addrs[j] ≠ ⊥ then
10:                 block ← (addr[j], leaf[j], data′)
11:                 Stash ← Stash ∪ block
12:     for j ← z to Z − 1 do
13:         read a random valid dummy

1:  function WriteBucket(bucket, Stash)
2:      find up to Z blocks from Stash that can reside
3:      in this bucket, to form addrs, leaves, data′
4:      ptrs ← PRP(0, Z + S)
5:      for j ← 0 to Z − 1 do
6:          data[ptrs[j]] ← data′[j]
7:      valids ← {1}^{Z+S}
8:      count ← 0
9:      encrypt addrs, leaves, ptrs, data
10:     write out count, valids, addrs, leaves, ptrs, data
```

G. Security Analysis

We will consider the security of ReadPath, EvictPath, and EarlyShuffle subroutines.

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 (through a PRP). 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 evicting is chosen statically, and is public (reverse-lexicographic order). ReadBucket always reads exactly Z blocks from random looking slots (as a result of the PRP). WriteBucket similarly writes $Z + S$ encrypted blocks in a data-independent fashion. □

Claim 3. EarlyShuffle *leaks no information.*

The frequency and to which buckets EarlyShuffle occur to is publicly known; i.e., 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. Claim 1, 2, and 3 shows that the subroutines are secure, and thus the security for Ring ORAM reduces to bounding the stash size, which we address in § V.

H. Optimizations

Several optimization techniques proposed for other ORAM constructions also apply to Ring ORAM. Of particular interest is the XOR technique proposed in Burst ORAM [4] and level compression proposed in [22]. Both optimizations are designed for hierarchical ORAM constructions, and are not compatible with other tree-based ORAMs. Since Ring ORAM buckets resemble the structure of hierarchical ORAMs, both the XOR technique and level compression apply to Ring ORAM. We remark that these two techniques are inherently incompatible with each other (even for hierarchical ORAM constructions). Of the two, the XOR technique is more interesting to us since it reduces online cost, which is essential in the Burst ORAM setting we evaluate in § VII-B. Below, we describe how to adopt the XOR technique to Ring ORAM.

XOR Technique. The key observation to enable the XOR technique is that except for the block of interest, all the other blocks returned by ReadPath are dummy blocks. To be more concrete, on each access ReadPath returns $L + 1$ blocks in ciphertext, one from each bucket, $\mathsf{Enc}(b_0), \mathsf{Enc}(b_2), \cdots, \mathsf{Enc}(b_L)$. With the XOR techique, ReadPath will return *a single ciphertext* — the ciphertext of all the blocks XORed together, namely $\mathsf{Enc}(b_0) \oplus \mathsf{Enc}(b_2) \oplus \cdots \oplus \mathsf{Enc}(b_L)$. The client can recover the encrypted block of interest by XORing the returned ciphertext with the encryptions of all the dummy blocks. Computing encrypted dummy blocks is trivial if the client sets the plaintext of all dummy blocks to a fixed value of its choosing (e.g., 0).

Tree Top Caching. We also remark that Ring ORAM can be tree top cached like Path ORAM [14]. The idea is simple: we can reduce the bandwidth for ReadPath and EvictPath by storing the top t Ring ORAM tree levels at the client as an extension of the stash. t is a parameter and for a given t, the stash grows by approximately $(2^t - 1)Z \approx 2^t Z$ blocks. As an example parameterization, setting $t = L/2$ makes the Ring ORAM or Path ORAM stash become $O(\sqrt{N})$ in size (which matches SSS ORAM [22]) and improves bandwidth by a factor of 2. As we will show in § VII-A, however, Path ORAM incurs $> 4\times$ more bandwidth than either Ring ORAM or Burst ORAM even when given this amount of storage.

I. 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 [20]: 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. Since recursion for Ring ORAM behaves in the same way as all the other tree-based ORAMs, we omit the details.

IV. Scheduling Operations

In this section, we describe methods for minimizing ORAM access response time, handling long bursts of client requests (as in Burst ORAM [4]) and minimizing the number of round trips for our construction.

A. De-amortizing Evictions

An interesting property of Ring ORAM is that it is asymmetric: its online cost (ReadPath, reads $L + 1$ blocks) is significantly lower than the worst case cost (EvictPath, touches $(2Z + S)(L + 1)$ blocks and causes Ring ORAM to go offline for some time). We can reduce the worst case cost in practice by de-amortizing a EvictPath operation across the *next* period of A ORAM accesses. The idea is to use any available idle bandwidth between requests to perform some amount of work for the previous EvictPath. Note that a delayed EvictPath operation will have at least as good eviction quality as a non-delayed one, because the WriteBucket steps have more blocks in the stash to choose from.

B. Bursts and Delayed Evictions

Dautrich el al. [4] consider the scenario where accesses to ORAM are bursty: e.g., millions of accesses are made in a short period, followed by a relatively long idle time where there are few requests. In this section, we propose a very simple scheme to make Ring ORAM good at handling bursty requests. The high level idea is to allow Ring ORAM to delay multiple (potentially millions of) EvictPath operations after the burst of requests. As the previous subsection pointed out,

delayed EvictPath can only improve eviction quality, leading to a smaller stash occupancy (after the delayed evictions finish). However, with the evenly scheduled EvictPath operations delayed, we will experience a much higher early reshuffle rate in levels towards the root. The solution is to coordinate tree top caching (§ III-H) with the maximum number of delayed evictions, which we describe below in detail.

Recall from § III-H the parameter t which means we cache the top t levels of the tree in client storage. For a given t, we allow at most 2^t EvictPath operations to be delayed, which means we can serve $2^t A$ ReadPath operations back to back. For any level $\geq t$, the early reshuffle rate stays the same: since a bucket at level t is touched once every 2^t EvictPath operations, delaying up to 2^t EvictPath operations does not affect that bucket. Tree top caching t levels require $2^t Z$ extra blocks in client storage, but notice that handling a burst length of $2^t A$ fundamentally requires that at least $2^t A$ blocks will accumulate in client storage anyway. In fact, our theoretical analysis shows that Z approaches $A/2$ (§ V-E), so the additional space for tree top caching is not a majority of the extra storage. To summerize, we can handle up to $2^t A$ requests in a burst, with only $2^t(A+Z)$ extra blocks in client storage.

C. Minimizing Rountrips

To keep the presentation simple, we wrote the ReadPath and EvictPath algorithms to process buckets one by one. While bandwidth is usually a bottleneck for ORAM, round-trip latency can be even more expensive in some settings like remote file servers (§ II-B). In this section, we describe how to minimize the number of round-trips for each operation in our scheme.

1) ReadPath: The entire ReadPath operation can be implemented in two round trips. The first round-trip executes GetBlockOffset for all buckets along the path, which involves reading metadata in each bucket. The second round-trip reads the correct offset to fetch one block from each bucket.

2) ReadBucket: ReadBucket can be implemented in no more than two round trips. The first round trip reads metadata to determine the offsets first, and the second one reads Z blocks from each bucket. For settings where roundtrip cost is really expensive, we can also read the entire $Z+S$ blocks plus the metadata in a single round trip, wasting some bandwidth. WriteBucket can be implemented in one round-trip.

3) EvictPath: It should now be obvious that EvictPath can be implemented in either two round-trips plus a write back. It first performs ReadBucket to all buckets on that path independently. After deciding what blocks each bucket should receive, it performs WriteBucket to all levels in a single round-trip.

4) EarlyReshuffle: EarlyReshuffle by itself also needs two or three round-trips. But we remark that EarlyReshuffle always follows ReadPath and can piggyback on ReadPath, which already spends two round-trips reading metadata and blocks. Thus, EarlyReshuffle only needs an extra round-trip for WriteBucket.

5) Augmented Position Map: After applying all the above optimizations, each ReadPath still requires two round trips. One way to achieve a single round-trip is to store more metadata in client storage.

The basic position map only tracks which path each block is mapped to. To perform a ReadPath in a single round trip, we also need to know (1) which bucket on the path contains the block of interest, (2) in which permuted slot of that bucket the block of interest lives, (3) which dummy slot in all other buckets should be touched.

We can augment the position map to include the first two pieces of information. I.e., the position map now stores a (l, i, j) triplet for each block, meaning that a block is in slot $\mathcal{P}(l, i, j)$. When EvictPath (or EarlyReshuffle) reshuffles a path (or a bucket), it needs to update these two fields of the position map for all affected blocks. It is worth pointing out that the augmented position map is *not* compatible with recursion for exactly this reason: in a recursive ORAM, each position map update requires multiple full ORAM accesses. Therefore, augmented position map only applies in the settings where client storage is not an issue.

For the third piece of required information, we store on the client side a dummycount. When a dummy block needs to be fetched from a bucket, we access the slot with offset $\mathsf{PRP}(\mathsf{EncSeed}, \mathsf{dummycount})$[3] and increment dummycount. We note that count for each bucket also needs to be stored in client storage, and dummycount is different from count. count tracks the number of times ReadPath touches a bucket since its last reshuffle, while dummycount tracks the number of times ReadPath fetches a *dummy* block from a bucket. EvictPath or EarlyReshuffle resets both dummycount and count, and also refreshes EncSeed.

Obviously, the level and offset fields are $\log L$ and $\log(Z+S)$ bits, respectively. dummycount and count are both $\log S$ bits, while the length of EncSeed is a security parameter λ (e.g. 128). The entire storage overhead for augmented position map is therefore position map plus bucket metadata, namely $N * (L + \log L + \log(Z+S))$ + $2^{L+1} * (2 \log S + \lambda)$ bits.

V. Stash Analysis

In this section we analyze the stash occupancy for a non-recursive Ring ORAM. Following the notations in Path ORAM [24], by $\mathsf{ORAM}_L^{Z,A}$ we denote a non-recursive Ring ORAM with $L+1$ levels and bucket size Z which does one RW access per A RO 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 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.

[3]We are reusing EncSeed to generate the offsets of dummy blocks.

A. Proof outline

The proof consists of the two steps. The first step is the same as Path ORAM. Our proof needs Lemma 1 and Lemma 2 in the Path ORAM paper [24], which we restate in § V-B. Lemma 1 introduces ∞-ORAM, which has a infinite bucket size and after the post-processing algorithm G has exactly the same distribution of blocks over all buckets and the stash. Lemma 2 says the stash usage of ∞-ORAM after post-processing is greater than R if and only if there exists a subtree T in ∞-ORAM whose "usage" exceeds its "capacity" by more than R. We note, however, that Path ORAM [24] only gives intuition for the proof of Lemma 1, and unfortunately does not capture of all subtleties. In Appendix § IX, we rigorously prove that lemma, which turns out to be quite tricky and requires significant changes to the post-processing algorithm.

The second step 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 average usage of subtrees in ∞-ORAM, and apply a Chernoff-like bound on their actual usage to complete the proof. We do not need to the complicated eviction game, negative association, stochastic dominance etc. in the Path ORAM proof [23].

B. ∞-ORAM

We first introduce ∞-ORAM, denoted as $\mathsf{ORAM}_L^{\infty,A}$. Its buckets have infinite capacity, i.e., an infinite Z. It receives the same input request sequence as a Ring ORAM. 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 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 [24]), 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[4], 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 $\mathcal{S}_\infty$ is post-processed to $\mathcal{S}_Z$, denoted by $G_{\mathcal{S}_Z}(\mathcal{S}_\infty) = \mathcal{S}_Z$, if no error occurs during G and b_i^∞ after G contains the same set of blocks as b_i^Z for $i = 0, 1, \cdots 2^{L+1}$.

[4]Note that the decreasing order ensures that a parent is always processed later than its children.

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

We leave the proof of Lemma 1 to Appendix § IX.

Next we investigate what state $\mathcal{S}_\infty$ will lead to the stash usage 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 the usage $X(T)$ 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 $\exists T \in \mathsf{ORAM}_L^{\infty,A}$ such that $X(T) > c(T) + R$ before post-processing.*

The proof of Lemma 2 remains unchanged from the Path ORAM paper [24]. For completeness, we restate the proof in Appendix § IX.

By Lemma 1 and Lemma 2, we have

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

C. Average Bucket and Rooted Subtree Load

The following lemma will be used in the next subsection:

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

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

Proof. 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 RW access to that leaf. Note that only real blocks could be put in b on that last access (stale blocks could not[5]), although some of them may have turned into stale blocks. 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 RW access to b's left child is the m_1-th RW access, and the last RW access to b's right child is the m_2-th RW access. 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

[5]Stale blocks can never be moved into a leaf by an RW access, because that RW access would remove all the stale blocks mapped to that leaf.

of RW accesses 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 $m_1 < m^* \leq m_2$ can be in b. There are at most $d = A|m_1 - m_2|$ such blocks [6] and each goes to b independently with a probability of $2^{-(i+1)}$, where i is the level of b. The deterministic nature of RW accesses in Ring ORAM makes it easy to show[7] that $|m_1 - m_2| = 2^i$. Therefore, $E[Y(b)] \leq d \cdot 2^{-(i+1)} = A/2$ for any non-leaf bucket as well. □

D. Bounding the Stash Size

$X(T) = \sum_i X_i(T)$, 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. Thus, we can apply a Chernoff-like bound [2] to get an exponentially decreasing bound on the tail distribution. To do so, let us first establish a bound on $E\left[e^{tX(T)}\right]$ where $t > 0$,

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

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

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

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

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

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 decrease exponentially in the stash size R,

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

[6] Only real or stale blocks with the right time stamp 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.

[7] One way to see this is that a bucket b at level i will be written every 2^i RW accesses, and two consecutive RW accesses to b always travel down the two different children of b.

TABLE IV: Maximum stash occupancy for realistic security parameters λ and several choices of A and Z. We can only achieve $Z = 16$,$A = 23$ in practice.

		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

E. Stash Size in Practice

Now that we have established that $Z\ln(2Z/A) + A/2 - Z - \ln 4 > 0$ ensures negligible 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 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 negligible stash overflow 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 slightly smaller A than we are able to achieve in practice, indicating that analysis is tight.

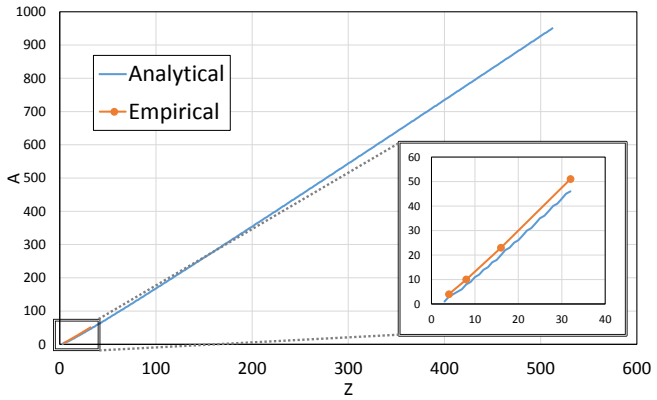


Fig. 3: For each Z, determine analytically and empirically the maximum A that results in negligible stash failure probability.

In Figure 4, we show a histogram of stash occupancies for several Z, A pairs. In all cases, histogram frequency decreases exponentially as stash occupancy increases. We extrapolate maximum stash size needed for a stash overflow probability of $2^{-\lambda}$ for several realistic λ in Table V. We show $Z = 16$, $A = 23$ for completeness: this is an aggressive setting that works for $Z = 16$ in practice but does not satisfy the theoretical analysis, It results in roughly $3\times$ the stash occupancy for a given λ.

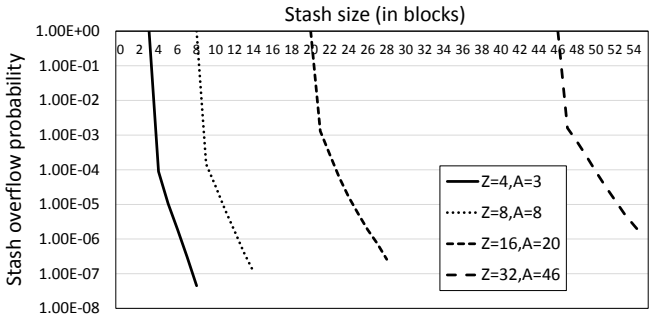


Fig. 4: For several Z A pairs, record the probability that the stash exceeds a certain number of blocks. The stash is sampled immediately after each EvictPath operation; hence the stash never contains $< A$ blocks.

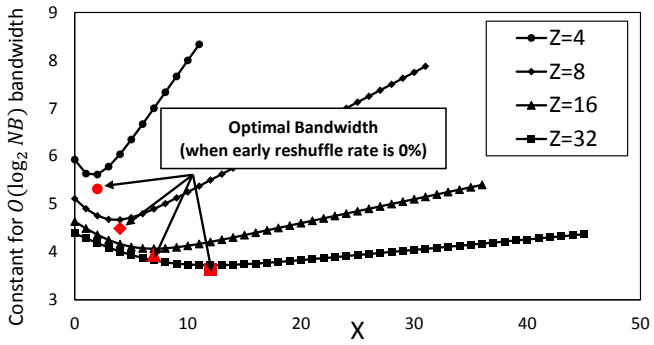


Fig. 5: For different Z, and the corresponding optimal A, vary S and plot bandwidth overhead. To normalize to different A, we sweep X where $S = A + X$.

VI. 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)? We first give a methodology for choosing A and S, to minimize bandwidth overhead, for a given Z. After that, we explore trade-offs in choosing Z.

A. How to Choose A and S Given Z

To begin with, we 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, the XOR technique, and level compression. 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 theoretic analysis (§ V) we have $L = \log(2N/A)$, so the ideal 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 (§ V-D).

Now we consider the extra overhead from early reshuffles (§ III-E). Recall from § III-F that when we process a bucket during an EvictPath or EarlyReshuffle, we read Z blocks and write back $Z+S$ blocks. Thus, 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. We show this effect through simulation in Figure 5: for S too small, early shuffle rate is high and bandwidth increases; for S too large, eviction bandwidth dominates. Diamonds in the figure show the bandwidth as if there were no early reshuffles We can see that with the optimal S, bandwidth overhead from early reshuffles is small.

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. This follows directly from the reverse-lexicographic order of eviction paths (§ III-D). 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 cumulative density function $\mathsf{Binom\_cdf}(S, 2^l A, 2^{-l})$[8]. This probability quickly converges a Poisson cumulative density function.

Now the overall bandwidth of Ring ORAM, 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 $(2Z+S)(1+\mathsf{Poiss\_cdf}(S,A))$. Not shown, this method always finds the optimal S and perfectly matches the overall bandwidth in our simulation in Figure 5.

We recap how to choose A and S for a given Z below. For the rest of the paper, we will choose A and S this way unless otherwise stated.

> 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 overall bandwidth:
> $\left(\frac{(2Z+S)(1+\mathsf{Poiss\_cdf}(S,A))}{A}+1\right)\log(4N/A)$

B. Trade-offs in Choosing Z

We now discuss bandwidth and client-storage trade-offs in choosing Z. In Figure 3, we observe that as Z increases, A/Z increases. In fact, the theoretic analysis in § V-D suggests that as $Z \to \infty$, A/Z increases and approaches 2. Then, since the bandwidth of EvictPath operations is $(2Z+S)/A$ and $S \approx A$

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

for large A, increasing Z ideally should improve bandwidth. Additionally, increasing Z by a factor of 2 roughly decreases L by 1 which further reduces bandwidth. This is indeed the case given sufficiently large block sizes, e.g. $B = \infty$ in Figure 6. For small block sizes (e.g., $B = 64$ Bytes), however, Figure 6 shows how increasing Z hurts bandwidth. In this regime, the overhead of accessing bucket metadata becomes the bottleneck and outweighs the benefit of larger A and Z.

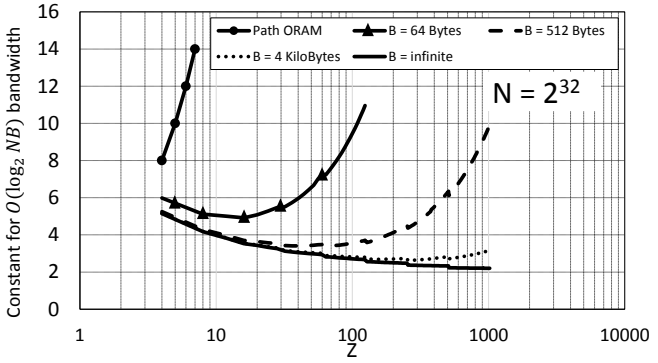


Fig. 6: Impact on overall bandwidth from increasing Z. All Ring ORAM points store bucket header metadata externally and $B = \infty$ approximates the case when the block size is large enough that reading bucket headers to contribute negligible additional bandwidth. A and S are set using the method in § VI-A. We show Path ORAM for reference and note that increasing Z for Path ORAM strictly increases bandwidth.

It is also worth pointing out that a larger Z increases the transient stash size because each EvictPath operations temporarily reads $\leq Z(L+1)$ blocks into the stash. Thus, choosing Z is a trade-off between stash size and eviction (offline) bandwidth, and depends on block size. In the next evaluation (§ VII-A), we explore which Z lead to an overall minimum bandwidth, given concrete client storage budgets and the costs of additional optimizations such as tree top caching.

VII. Evaluation

In this section, we compare Ring ORAM to the prior-art schemes in the large client storage (SSS ORAM [4], [22]) and small client storage (Path ORAM [24]) setting.

A. Overall Bandwidth vs. Client Storage

Our first study compares Ring ORAM, Path ORAM [24] and SSS ORAM [22] in terms of total bandwidth overhead (relative to accessing a block without ORAM) as a function of the client's local storage budget. All schemes are non-concurrent (i.e., without delayed/de-amortized evictions, § IV). A summary of competitive parameters is given in Table V.

1) Large client storage: Figure 7 shows bandwidth vs. client storage in the large client storage setting (e.g., remote file servers, § II-B). We show data for $N = 2^{28}$ and $B = 256$ KBytes (for a 64 TeraByte ORAM) to match the results in Figure 9 of [22] and remark that with large client storage, the trade-offs are similar for different block sizes and capacities. For all schemes, we account for all sources of client storage including the position map, extra metadata, shuffle

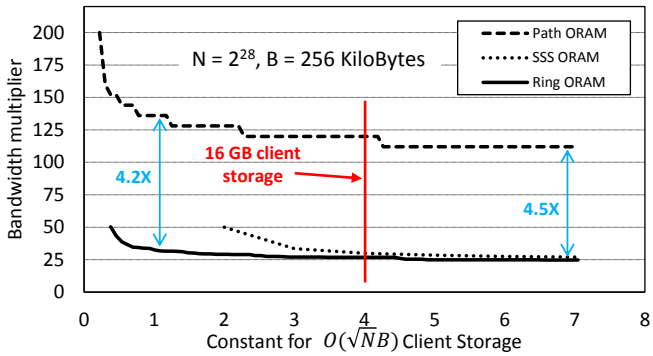


Fig. 7: Bandwidth overhead vs. client storage given large client storage (i.e., appropriate in the remote file server setting).

buffer (SSS ORAM only) and stash for a security parameter of $\lambda = 80$. Ring ORAM uses the augmented position map from § IV-C5.

For each scheme, 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 greedily tree top cache as much of the ORAM tree as possible. For Ring ORAM we simultaneously adjust Z and the amount of tree top caching (§ III-H) to minimize bandwidth for a given storage. We then select A and S based on the method in § VI-A, and apply the XOR technique (§ III-H). For $A > 1500$, we set $S = A + 50$ as this is sufficient to achieve a negligible early reshuffle rate. The SSS ORAM uses level compression and adjusts its eviction rate to fill additional space.[9]

We plot each scheme starting at the minimum required client storage, given the position map. In Figure 7, for client storage $4\sqrt{N} * B = 16$ GB (the suggested setting from [22]), Ring ORAM achieves a $4.47\times$ bandwidth reduction relative to Path ORAM and an $31.2/26.8 = 16\%$ improvement relative to the SSS construction. For client storage $2\sqrt{N}$ blocks (the minimum space needed by the SSS construction), Ring ORAM outperforms the SSS ORAM by 72%. We note that the SSS ORAM uses level caching and therefore has a significantly larger online bandwidth than Ring ORAM. By instead comparing against the SSS ORAM with the XOR technique at $4\sqrt{N} * B$ storage, the SSS scheme's online bandwidth is comparable to Ring ORAM but Ring ORAM gains a $35.7/26.8 = 33\%$ advantage in overall bandwidth.

2) Small client storage: Figure 8 shows bandwidth vs. client storage in the small client storage setting (e.g., secure processors, § II-C). All configurations assume a 1 TeraByte ORAM capacity. This study does not include the position map or the cost of recursion (§ III-I) in the client storage or bandwidth as this effect will be similar for both Path ORAM and Ring ORAM. (We study the effect of recursion in § VII-C, where we evaluate Ring ORAM in the secure

[9]We found applying level compression yields a better overall bandwidth relative to the XOR technique for the SSS ORAM (which is consistent with [4]). For completeness, we show the SSS scheme with the XOR technique in Table V.

TABLE V: Highlighted parameter settings for different scenarios, optimizing for overall bandwidth given concrete client storage budgets (§ VII-A). Overheads are relative to an insecure system. Ring is Ring ORAM (this paper), SSS is the scheme from [22] and Path is Path ORAM [24]. XOR means we apply the XOR technique [4] and LC means we apply level compression [22]. Parameter meaning is given in Table II.

Ring ORAM parameters						Online, Overall Bandwidth (relative to insecure)				
N	B	Z	A	S	Client storage	**Ring**	**Ring (XOR)**	SSS (LC)	SSS (XOR)	Path
Large client storage (ORAM capacity = 64 TeraBytes)										
2^{28}	256 KByte	742	1395	1445	16 GBytes	13×, 39.9×	1×, 26.8×	7.8×, 31.2×	1×, 35.7×	60×, 120×
Small client storage (ORAM capacity = 1 TeraByte)										
2^{34}	64 Byte	10	11	16	73 KBytes	48.1×, 137×	24.1×, 113×	N/A		117×, 235×
2^{31}	512 Byte	21	28	37	489 KBytes	27.9×, 94.5×	5.9×, 71.9×	N/A		93×, 186×
2^{28}	4 KByte	17	22	29	3.1 MBytes	20.4×, 79.3×	1.4×, 59.7×	N/A		80×, 160×

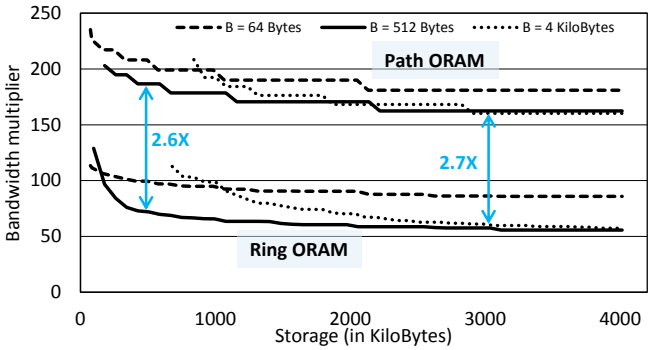


Fig. 8: Bandwidth overhead vs. client storage given small client storage (i.e., appropriate in the secure processor setting).

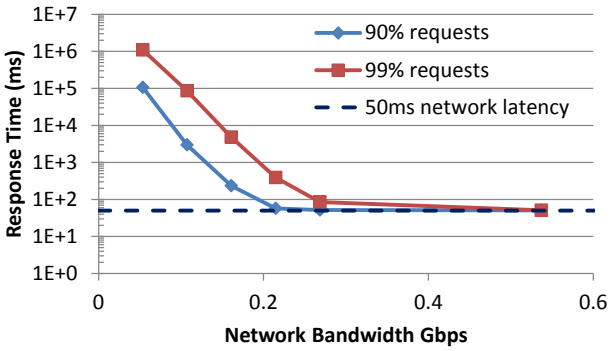


Fig. 9: Response time of a baseline system (without ORAM).

processor setting.) To make Ring ORAM compatible with recursion, however, we do not use the augmented position map from § IV-C5. Otherwise, we use the same optimizations as stated in § VII-A1.

For different block sizes, Ring ORAM uses different strategies to improve bandwidth. For the 64 Byte block size, Ring ORAM must use a small Z to minimize header bandwidth and thus bandwidth improves slowly — a function of tree top caching only. For larger block sizes and sufficient storage, Ring ORAM quickly improves by increasing Z.

In Table V, we highlight our bandwidth overhead given $\log^2 N{*}B$ blocks of client storage. This budget is large enough for Ring ORAM to decrease bandwidth by increasing Z yet small enough for practical scenarios such as secure processors. At this setting, the 64 Byte, 512 Byte and 4 KByte block size configurations require 73 KBytes, 489 KBytes and 3.1 MBytes of client storage and improve over Path ORAM by 2.1×, 2.6× and 2.7×, respectively. For reference, these storage budgets are between .4-2.3% of $1.75\sqrt{N} * B$ (the minimum budget for the SSS construction in § VII-A1, when the position map is excluded). By increasing storage to $1.75\sqrt{N}{*}B$, Ring ORAM overhead (and improvement over Path ORAM) for 512 Byte and 4 KByte blocks becomes 43× (2.9×) and 32× (3.7×).

We remark that in some cases, the XOR technique may not be appropriate in the small client storage setting because memory modules such as DRAM cannot perform computation (§ II-C). For completeness, we give our results without the XOR technique in Table V.

B. Case Study: Bursty File-Server Workloads

We now study Ring ORAM's online bandwidth and ability to handle bursts of requests using an enterprise trace of requests to a remote file server. We use the same trace and assume the same experimental setup and metric as the Burst ORAM paper [4] after private correspondence with the authors [4]. Network latency is fixed to be 50 ms, and we experiment under different network bandwidth. The metric of merit is percentile response time. A 90 (or 99) percentile response time of T means 90% (or 99%) of the requests complete in T. We first measure and report the percentile response time of a baseline system without ORAM in Figure 9. Generally, when network bandwidth is too low we see huge response time because requests come in faster than the rate they can be processed and thus they pile up. As bandwidth increases, response time approaches the ideal network latency of 50 ms.

We now compare Ring ORAM with Burst ORAM, giving both schemes 100 GByte as done in [4]. For Ring ORAM, we use the techniques to handle bursts (§ IV-B) and minimize round-trips (§ IV-C), including augmented position map (§ IV-C5). We set $Z = 90$ and $A = 150$ using the method in § VI-A. Since the metric of merit is online bandwidth, increasing Z and A further has little benefit. However, we set $S = 220$ which is larger than the analysis-recommended $S = 174$, again because the metric of merit is minimum online bandwidth: we wish to minimize the cost of EarlyReshuffle, even if that increases the cost of EvictPath.

In Figure 10, we report Ring ORAM and Burst ORAM percentile response time relative to the baseline (no ORAM).

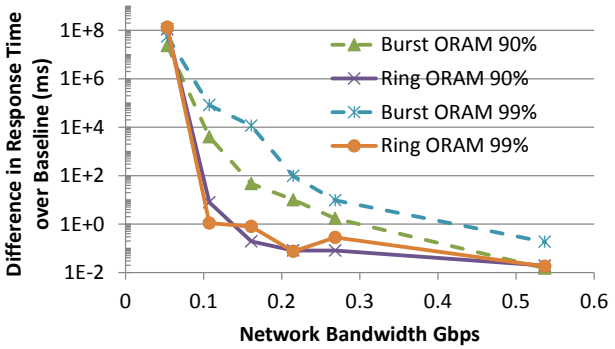


Fig. 10: Difference in response time over baseline of Burst ORAM [4] and Ring ORAM.

The Burst ORAM numbers come directly from Figure 11 of that work [4]. With limited network bandwidth (< 0.05 Gbps), both schemes experience very long response times due to the bandwidth overhead of ORAM. With ample bandwidth (> 0.35 Gbps), both ORAMs introduce negligible overhead. With medium network bandwidth, Ring ORAM shows a significant advantage over Burst ORAM. Ring ORAM increases percentile response time by less than 1 ms, while Burst ORAM often sees a 10^2 to 10^4 ms increase in percentile response time. We believe that our improvement is due to Burst ORAM's early cache-in scheme, which increases their online bandwidth, causing them to fall behind in requests and also takes up extra slots from their stash, affecting their ability to handle long bursts. On the other hand, we do not use early cache-in and parameterize tree top caching and S to minimize early reshuffle rate (§ IV-B).

C. Case Study: Secure Processors

In this study, we show how Ring ORAM improves the performance of secure processors. We assume the same processor architecture as [18], given in Table 4 of that work. That is, we evaluate a 4 GByte ORAM with 64-Byte block size (matching a typical processor's cache line size), and use the standard ORAM recursion technique with 32-Byte block size for position map ORAMs (§ III-I). Due to the small block size, we parameterize Ring ORAM at $Z = 5, A = 5, X = 2$ to reduce metadata overhead. We give both ORAMs up to 256 KBytes for the final position map, which requires applying recursion 3 times. We do not use tree top caching for any scheme (as this proportionally benefits both Ring ORAM and Path ORAM) or the XOR technique (as current DRAM DIMMs do not have the ability to perform computation).

We evaluate performance for SPEC-int benchmarks and two database benchmarks, and simulate 3 billion instructions for each benchmark. Figure 11 shows program slowdown using Path ORAM and Ring ORAM over an insecure DRAM. Ring ORAM with de-amortized eviction offers significantly better performance: most benchmarks see less than $3\times$ slowdown, with the geometric average slowdown being $2.2\times$. This is $2.75\times$ speedup over Path ORAM on average.

Figure 12 shows the average response time of ORAM for

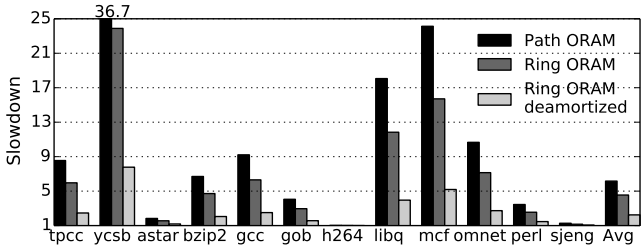


Fig. 11: SPEC benchmark slowdown.

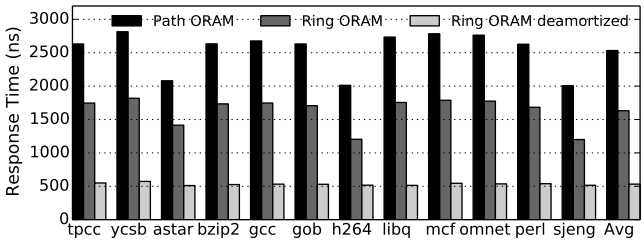


Fig. 12: SPEC benchmark response time.

each benchmark. Shorter response time means programs are blocked less by ORAM (which is a function of response time, i.e. ReadPath, and even EvictPath if the ORAM receives more requests than it can handle in a given eviction period). The de-amortized Ring ORAM utilizes idle time in memory (when programs don't need memory) to hide the expensive eviction operation, effectively reducing response time by more than 50% compared to Ring ORAM without de-amortization. Compared to Path ORAM, the response time improvement of the de-amortized Ring ORAM is almost $5\times$.

VIII. Conclusion

This paper proposed and analyzed Ring ORAM, the most bandwidth-efficient ORAM scheme for both the large and small (poly-log) client storage setting. Ring ORAM is simple, flexible and competitive in bandwidth with multiple prior-art schemes in different settings — simply by tuning parameters.

We show that Ring ORAM improves overall bandwidth by $2 - 2.7\times$ and online bandwidth by $> 50\times$ relative to Path ORAM — the prior-art poly-log storage scheme. Given large client storage, Ring ORAM Ring ORAM improves upon Path ORAM in bandwidth (given equal storage) by $4.5\times$ and the SSS ORAM by 16-33% — the prior-art scheme for large storage. Using a realistic enterprise file system trace and remote secure processor workloads, we further show that Ring ORAM exceeds prior-art work in their respective setting.

REFERENCES

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

[2] H. Chernoff et al. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *The Annals of Mathematical Statistics*, 23(4):493–507, 1952.

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

[4] J. Dautrich, E. Stefanov, and E. Shi. Burst oram: Minimizing oram response times for bursty access patterns. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 749–764, San Diego, CA, Aug. 2014. USENIX Association.

[5] C. Fletcher, M. van Dijk, and S. Devadas. Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *Proceedings of the 7th ACM CCS Workshop on Scalable Trusted Computing; an extended version is located at http://csg.csail.mit.edu/pubs/memos/Memo508/memo508.pdf (Master's thesis)*, pages 3–8, Oct. 2012.

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

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

[8] O. Goldreich and R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[9] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In *Proceedings of the 38th International Conference on Automata, Languages and Programming - Volume Part II*, ICALP'11, pages 576–587, Berlin, Heidelberg, 2011. Springer-Verlag.

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

[11] M. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[12] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in) security of hash-based oblivious ram and a new balancing scheme. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 143–156. SIAM, 2012.

[13] J. R. Lorch, B. Parno, J. W. Mickens, M. Raykova, and J. Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 199–213, San Jose, CA, 2013. USENIX.

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

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

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

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

[18] L. Ren, C. Fletcher, X. Yu, A. Kwon, M. van Dijk, and S. Devadas. Unified oblivious-ram: Improving recursive oram with locality and pseudorandomness. Cryptology ePrint Archive, Report 2014/205, 2014.

[19] L. Ren, X. Yu, C. Fletcher, M. van Dijk, and S. Devadas. Design space exploration and optimization of path oblivious ram in secure processors. In *Proceedings of the Int'l Symposium on Computer Architecture*, June 2013. Available at Cryptology ePrint Archive, Report 2013/76.

[20] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious ram with $o((\log n)^3)$ worst-case cost. In *Asiacrypt*, pages 197–214, 2011.

[21] E. Stefanov and E. Shi. Oblivistore: High performance oblivious cloud storage. In *Proc. of IEEE Symposium on Security and Privacy*, 2013.

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

[23] E. Stefanov, M. van Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. ACM CCS, 2013. Available at Cryptology ePrint Archive, Report 2013/280.

[24] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. In *Proceedings of the ACM Computer and Communication Security Conference*, 2013.

[25] P. Williams and R. Sion. Single round access privacy on outsourced storage. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 293–304, New York, NY, USA, 2012. ACM.

[26] P. Williams, R. Sion, and A. Tomescu. Privatefs: A parallel oblivious file system. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012.

[27] X. Yu, C. W. Fletcher, L. Ren, M. van Dijk, and S. Devadas. Generalized external interaction with tamper-resistant hardware with bounded information leakage. In *Proceedings of the Cloud Computing Security Workshop (CCSW)*, 2013.

[28] X. Zhuang, T. Zhang, and S. Pande. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. In *Proceedings of the 11th ASPLOS*, 2004.

IX. APPENDIX: PROOF OF LEMMAS

To prove lemma 1, we made a little change to Ring ORAM algorithm. In Ring ORAM, an RO access 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 RW access 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 access (either RO or RW). An RO access 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 RW access. Let RW_l^Z be an RW access to $\mathcal{P}(l)$ (path l) in $\mathsf{ORAM}_L^{Z,A}$ and RW_l^∞ be an RW access to $\mathcal{P}(l)$ in $\mathsf{ORAM}_L^{\infty,A}$. Then, $\mathcal{S}'_Z = \mathsf{RW}_l^Z(\mathcal{S}_Z)$ and $\mathcal{S}'_\infty = \mathsf{RW}_l^\infty(\mathcal{S}_\infty)$. Note that RW_l^Z has the same effect as RW_l^∞ followed by post-processing, so

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

The last equation is due to the induction hypothesis.

It remains to show that

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

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 show that for any $0 < i < 2^{L+1}$, $G_{\mathcal{S}'_Z}(\mathsf{RW}_l^\infty(g_1 g_2 \cdots g_i(\mathcal{S}_\infty))) = G_{\mathcal{S}'_Z}(\mathsf{RW}_l^\infty(g_1 g_2 \cdots g_{i-1}(\mathcal{S}_\infty)))$. This is obvious if we consider the following three cases separately:

1) If $b_i \in \mathcal{P}(l)$, then g_i before RW_l^∞ has no effect since RW_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 RW_l^∞ touch non-overlapping buckets and do not interfere with each other. Hence, their order can be swapped, $G_{\mathcal{S}'_Z}\left(\mathsf{RW}_l^\infty\left(g_0 g_1 g_2 \cdots g_i\left(\mathcal{S}_\infty\right)\right)\right) = G_{\mathcal{S}'_Z} g_i\left(\mathsf{RW}_l^\infty\left(g_0 g_1 g_2 \cdots g_{i-1}\left(\mathcal{S}_\infty\right)\right)\right)$. Furthermore, $b_i^Z = b_i'^Z$ (since RW_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 RW_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 RW_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

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

and completes the proof. □

The proof for lemma 2 is much simpler. We replicate a version of the proof from [24] here.

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 $X(T) > c(T) + R$. □