Tiny ORAM: A Low-Latency, Low-Area Hardware ORAM Controller with Integrity Verification

Christopher W. Fletcher[†*], Ling Ren[†*], Albert Kwon[†], Marten van Dijk[‡], Emil Stefanov[○], Srinivas Devadas[†]

[†] Massachusetts Institute of Technology − {cwfletch, renling, kwonal, devadas}@mit.edu
[‡] University of Connecticut − vandijk@engr.uconn.edu
[○] University of California, Berkeley − emil@berkeley.edu
[∗] Christopher Fletcher and Ling Ren contributed equally to this work

July 22, 2014

Abstract

We propose and build *Tiny ORAM*, an ORAM construction that improves the state of the art Path ORAM in several dimensions.

First, through a construction that we call *RAW Path ORAM*, we reduce the number of symmetric encryption operations by 4× compared with Path ORAM. Raw Path ORAM also dramatically simplifies the theoretical analysis on the client's storage requirement (stash size). Second, we propose an integrity verification scheme that is asymptotically more efficient than prior work for position-based ORAMs. Third, through a construction that we call *Unified Path ORAM*, we reduce the empirical overhead of the recursive ORAM construction.

We demonstrate and evaluate a working prototype on a stock FPGA board. Of independent interest, Tiny ORAM is the first hardware ORAM design to support small client storage and arbitrary block sizes (e.g., 64 Bytes to 4096 Bytes). Block size flexibility allows Tiny ORAM to greatly reduce the worst-case access latency for ORAM running programs with erratic data locality. Tiny ORAM is also the first design to implement and report real numbers for the cost of symmetric encryption in hardware ORAM constructions. Tiny ORAM requires 3%/14% of the FPGA logic/memory (including the cost of encryption) and can complete an ORAM access for a 64 Byte block in $1.25 - 4.75 \mu s$.

Contents

1 Introduction **2**
1.1 Contributions 3

2 Threat Model **4**

3 Background **5**
3.1 Basic Path ORAM 5
3.1.1 Path ORAM Security 7
3.2 Recursive Path ORAM 7

4 Stash Management **8**
4.1 PushToLeaf With Bit Hacks 8
4.2 Hardware Implementation 9

5 Unified Path ORAM 10
5.1 PosMap Lookaside Buffer (PLB) . . . 10
5.2 Unified ORAM . . . 11
5.3 Security . . . 12
5.4 Generality of Construction . . . 12

6 RAW Path ORAM 12
6.1 Overview . . . 12
6.2 Path Write Predictability . . . 13
6.3 Security . . . 14
6.4 AES Area Savings In Practice . . . 14

7 Integrity Verification 15
7.1 Detailed Description . . . 15
7.2 Key Advantage: Hash Bandwidth and Parallelism . . . 16
7.3 Subtly Broken Schemes . . . 17
7.4 Other Overheads . . . 18
7.4.1 PosMap Storage . . . 18
7.4.2 MAC and Initialization Vector Storage . . . 18
7.5 Security . . . 18
7.5.1 Integrity . . . 19
7.5.2 Privacy . . . 19
7.6 Generality of Construction . . . 21

8 Evaluation 21
8.1 Evaluation Metrics . . . 21
8.2 Implementation . . . 21
8.3 Access Latency Comparison . . . 22
8.4 Hardware Area Comparison . . . 23

9 Related Work 23

10 Conclusion 24

A Unified ORAM Pseudo-Code 26

B RAW ORAM pseudo-Code 27

C RAW ORAM Stash Analysis 27
C.1 ∞-ORAM . . . 27
C.2 Average Subtree and Bucket Load . . . 29
C.3 Chernoff Bound . . . 30
C.4 Extension: Dummy Accesses . . . 30

1 Introduction

With cloud computing becoming increasingly popular, privacy of users' sensitive data has become a huge concern in computation outsourcing. In an ideal setting, users would like to "throw their encrypted data over the wall" to a cloud service that performs computation on that data, yet cannot learn any information from within that data. It is well known, however, that encryption is not enough to get privacy. A program's memory access pattern has been shown to reveal a large percentage of its behavior [34] or the encrypted data it is computing upon [16, 20].

Oblivious RAM (ORAM) is a cryptographic primitive that *completely* eliminates the information leakage in a program's memory access trace (made up of reads/writes to memory). Conceptually, ORAM works by maintaining all of memory in encrypted and shuffled form. On each access, memory is read and then reshuffled. Under ORAM, any memory access pattern is computationally indistinguishable from any other access pattern of the same length. ORAM was first proposed by Goldreich and Ostrovsky [11, 12], and there has been significant follow-up work that has resulted in more efficient and cryptographically-secure ORAM schemes [22, 21, 6, 3, 15, 17, 32, 27, 30].

An important use case for ORAM is in *trusted hardware* [20, 28, 30, 25, 8]. In this setting, ORAM client logic (called the *ORAM controller*) runs on a cloud processor which negotiates with an untrusted external memory such as disk, DRAM or flash. Unlike in the traditional file-server setting, ORAM controller logic in this setting is implemented directly in hardware. As silicon on a chip is limited, these designs are area constrained.

Recently Maas et al. [20] implemented *Phantom*, the first hardware ORAM prototype for the trusted hardware setting. Their design identifies several challenges for future hardware ORAM proposals, which we address in this paper.

The first challenge is that it is hard to design a hardware ORAM controller that supports small block sizes. Primarily to ensure small on-chip storage and high memory throughput, Phantom adopts 4-KiloByte blocks. While benefiting applications with good data locality, this large block size runs the risk that some data will never be used, which can be viewed as wasted bandwidth and energy consumption. More importantly, the latency to return a word from ORAM grows with block size. Indeed, most modern processors have a 64-Byte cache block size to reduce this waste and latency. One goal in this paper is to develop schemes that flexibly support any block size such that a designer can choose a suitable block size according to the needs of the application.

The second challenge is that, even when the ORAM controller's on-chip storage is small, it is hard to build hardware ORAM controllers whose overall chip area is small. Prior art hardware-implementable ORAM protocols require the same bandwidth for memory as they do for cryptographic operations, such as symmetric encryption. To prevent performance loss, many encryption units are needed which imposes large area overheads. For example, Phantom projects that AES units alone would take $\sim 50\%$ of the logic of a state-of-the-art field programmable gate array (FPGA) device. A second goal in this paper is to develop new ORAM schemes that reduce the required encryption bandwidth, and to carefully engineer the system to turn that theoretical savings into concrete hardware area reduction.

A third line of investigation that Phantom motivates but does not implement is integrity verification. Integrity verification is an important consideration for any secure storage system under active adversaries. We found, however, that the existing scheme based on Merkle trees [24], when implemented in hardware, once again exacerbates design area problems and leads to performance bottlenecks, when memory bandwidth is high and the block size is small. We will address this issue with a new, cheap in area and efficient integrity scheme for ORAM.

1.1 Contributions

In this paper, we present *Tiny ORAM*, a complete hardware ORAM controller prototype. Through novel algorithmic improvements and careful hardware design, Tiny ORAM enables (1) small blocks to unlock low latency, (2) scalability to large working sets, and (3) memory integrity checking. Despite these features, Tiny ORAM synthesizes to hardware with an extremely small hardware area footprint. We make a number of contributions listed below:

Bit-based block push-back to enable small blocks (§ 4). We develop a new block push-back scheme using efficient bit tricks that, when implemented in hardware, removes the stash eviction performance bottleneck from prior work [20]. Our scheme can support any reasonable block size (e.g., from 64-4096 Bytes) without impacting system performance.

***Unified ORAM* to efficiently provide working set scalability (§ 5).** For the first time, we build the recursive ORAM construction of Shi et al. [27] into hardware. We then propose a new *Unified ORAM* scheme that reduces the overhead of the recursive scheme by up to 39% on real workloads, when compared to a baseline Path ORAM [30] design that uses recursion.

***RAW ORAM* to reduce encryption bandwidth and simplify stash analysis (§ 6).** Inspired by Gentry et al. [9], we propose a new ORAM tool called *path write predictability (PWP)*, and use it to construct a new type of ORAM scheme called *RAW ORAM*. Compared to the baseline Path ORAM design, RAW ORAM requires $\sim 4\times$ fewer encryption units and maintains comparable bandwidth relative to Path ORAM. We further prove that RAW Path ORAM has negligible stash overflow probability under certain configurations (Appendix C). Interestingly, the analysis for RAW Path ORAM is much simpler than that for Path ORAM.

Integrity verification. We introduce a novel, simple and efficient verification scheme for position-based ORAMs (§ 7). Our scheme is asymptotically more efficient than prior schemes in that it requires $O(1)$ blocks be integrity checked/verified per non-recursive ORAM access (where each non-recursive ORAM access transfers $O(\log N)$ blocks).

Real implementation and evaluation (§ 8). We implement the ideas proposed above in hardware, with real hashing and encryption units, and evaluate our design for performance and area on a Virtex-7 VC707 FPGA board. Throughout the paper, we compare our design to Path ORAM [30], which we also implemented in hardware, and discuss various real-world hardware optimizations to make our theoretical gains translate to practice. With the VC707 board's 12.8 GB/s DRAM bandwidth, Tiny ORAM can complete an access for a 64 Byte block in $1.25-4.75\mu s$, depending on program address locality. This design requires 3% of the FPGA's logic and 14% of its on-chip memory. Adding integrity increases the logic/memory area total to 5%/15% of the FPGA. Our design is written entirely in Verilog-2001 with no proprietary components.

2 Threat Model

In our setting, trusted hardware (e.g., a secure processor) operates in an untrusted environment (e.g., a data center) on behalf of a trusted client. The processor runs a private or public program on private data submitted by the user, and interacts with a trusted on-chip ORAM controller, on last-level cache misses, to read/write data to untrusted external memory. We assume untrusted memory is implemented in DRAM for the rest of the paper.

The data center is treated as both a passive and active adversary. First, the data center will passively observe how the processor interacts with DRAM to learn information about the user's encrypted data. Second, it may additionally try to tamper with the contents of DRAM to influence the outcome of the program.

Security definition (privacy). We adopt the a slightly different definition for ORAM that is more compatible with trusted hardware and is assumed explicitly or is implicit in all prior hardware proposals [20, 25, 7]:

For data request sequence $\overleftarrow{S}$*, let* $\mathsf{ORAM}(\overleftarrow{S})$ *be the resulting randomized data request sequence of an ORAM algorithm that is visible to an adversary (data center). Each element in a data request sequences follows the standard RAM interface, i.e., is an (address, op, write data) tuple. We require that for any* $\overleftarrow{S}$, $\mathsf{ORAM}(\overleftarrow{S})$ *is computationally indistinguishable from* $\mathsf{ORAM}(\overleftarrow{S'})$*, for any* $\overleftarrow{S'}$*, if* $|\mathsf{ORAM}(\overleftarrow{S})| = |\mathsf{ORAM}(\overleftarrow{S'})|$.

The standard definition (mostly used in theoretical works) is that if $|\overleftarrow{S}| = |\overleftarrow{S'}|$, the resulting ORAM sequences should be indistinguishable. This usually means that $|\mathsf{ORAM}(\overleftarrow{S})|$ is completely determined (and thus revealed) by $\overleftarrow{S}$. However, this guarantee is not very useful in the trusted processor setting. Processors have several (usually 3 for modern processors) levels of on-chip cache. When a program requests data, the processor first looks for the data in its Level-1 (L1) cache; on an L1 miss, it accesses the L2 cache, and so on. Only when it misses all of the on-chip cache will the processor access the external memory. Whether it's a cache hit or miss depends on the actual access pattern. Thus, if $\overleftarrow{S}$ is the sequence of load/store instructions in a program, only a *data-dependent* fraction of them will be seen by ORAM. Satisfying the original ORAM definition requires completely disabling processor cache, which is impractical.

With our definition, we allow processors to use cache. As a result, $|\mathsf{ORAM}(\overleftarrow{S})|$ is now determined by, and thus reveals, the number of Last-Level-Cache (LLC) misses in $\overleftarrow{S}$, but not $|\overleftarrow{S}|$. If an adversary knows $|\overleftarrow{S}|$, it further learns the number of hits in cache. Both definitions capture the essence of ORAM's privacy guarantee: ORAM hides individual elements in the data request sequence, while leaking a small amount of information on the length of the sequence. From an information-theoretic point of view, the former grows linearly with the request sequence length, while the latter only grows logarithmically.

Security definition (integrity). *An ORAM provides integrity if it behaves like a valid memory with overwhelming probability from the processor's perspective. Memory has valid behaviors if the value the processor reads from a particular address is the most recent value that it has written to that address.*

Timing. We will design the ORAM controller such that each ORAM access is *atomic* from a timing perspective. By atomic, we wish for each DRAM request made *during* an ORAM access to occur at data-independent times. We do not obfuscate *when* an ORAM access starts or the time it takes the program to terminate. These two timing channels have been addressed for Path ORAM by Fletcher et al. [7] and that work can be applied on top of this work.

Physical attacks: Power, EM, RF, etc. Also in keeping with prior ORAM work, we don't consider leakage over the processor power pins, RF signals given off by the processor, or other channels that can only be observed through probes.

3 Background

As did Phantom, Tiny ORAM originates from Path ORAM [30]; we extend Path ORAM functionality in this paper. We now explain Path ORAM and the recursive ORAM construction in detail. Parameters and notations are summarized in Table 1.

3.1 Basic Path ORAM

Path ORAM organizes untrusted external DRAM as a binary tree which we refer to as the *ORAM tree*. The root node of the ORAM tree is referred to as level 0, and the leaf nodes as level L. We denote each leaf node with a unique leaf label l for $0 \leq l \leq 2^L - 1$. We refer to the list of buckets on the path from the root to leaf l as $\mathcal{P}(l)$.

Each node in the tree is called a *bucket* and can hold up to a small constant number of blocks denoted Z (typically $Z = 4$ to 5). We denote the block size in bits as B. In this paper, each block is a processor cache line (and we correspondingly set $B = 512$). Buckets that have less than Z blocks are padded with *dummy blocks*. Each bucket is encrypted using symmetric probabilistic encryption (e.g., AES in counter mode). Thus, an observer cannot distinguish real blocks from dummy blocks.

The Path ORAM controller (trusted hardware) contains a *position map*, a *stash* and associated control logic. The position map (PosMap for short) is a lookup table that associates each data block's logical address with a leaf in the ORAM tree. The stash (henceforth called Stash) is a random access memory (e.g., an SRAM) that stores up to a small number of data blocks. The stash capacity is $C + L * Z$ blocks. We say that the ORAM stash has overflowed if, at the beginning of an ORAM access, the number of blocks in the stash is $> C$ (§ 3.1.1). To achieve negligible stash overflow probability for real security parameters, $C = 50$ to 100 is typical. Together, the PosMap and Stash make up Path ORAM's client storage (from § 1).

Path ORAM Invariant. At any time, each data block in Path ORAM is mapped to a random leaf via the PosMap. Path ORAM maintains the following invariant: If a block is mapped to leaf l, then it must be either in some bucket in $\mathcal{P}(l)$ or in Stash.

Algorithm 1 shows how Path ORAM read/writes a block with program address a. We split this algorithm into a pair of algorithms, called $\mathsf{PORAMFrontend}()$ and $\mathsf{PORAMBackend}()$, which will simplify the presentation later on. The frontend operation looks up the PosMap to determine the leaf l for address a (Lines 6-8). The backend operation first reads/decrypts the buckets on path l into Stash (Lines 22-24). All encryption operations ($\mathsf{decrypt}()$ and $\mathsf{encrypt}()$) use a session key K controlled by the trusted processor. The

Table 1: ORAM parameters and notations.

Notation	Meaning
L	Depth of Path ORAM tree
Z	Data blocks per ORAM tree bucket
N	Number of real data blocks in tree
B	Data block size (in bits)
C	Stash capacity (in blocks, not including transient storage)
K	Session key (controlled by trusted processor)
$\mathcal{P}(l)$	Path to leaf l in ORAM tree
$\mathcal{P}(l)[i]$	i-th bucket on Path $\mathcal{P}(l)$
Recursive ORAM (§ 3.2) only	
X	Number of leaf labels in a PosMap block
H	Number of ORAM recursion levels
RAW ORAM (§ 6) only	
A	The number of RO accesses per RW access

Algorithm 1 Basic Path ORAM access.

```
Inputs: Address $a$, Operation $op$, Write Data $D'$
function AccessORAM($a, op, D'$)
    $l, l' \leftarrow$ PORAMFrontend($a$)
    return PORAMBackend($a, l, l', op, D'$)
function PORAMFrontend($a$)
    $l' \leftarrow \mathsf{PRNG}_K() \mod 2^L$
    $l \leftarrow$ PosMap[$a$]
    PosMap[$a$] $\leftarrow l'$                                  ▷ remap block
    return $l, l'$
function PORAMBackend($a, l, l', op, D'$)
    ReadPath($l$)
    $r \leftarrow$ FindBlock(Stash, $a$)                         ▷ $r$ points to block $a$
    $(a, l, D) \leftarrow$ Stash[$r$]
    if $op \overset{?}{=}$ write then
        Stash[$r$] $\leftarrow (a, l', D')$
    else if $op \overset{?}{=}$ read then
        Stash[$r$] $\leftarrow (a, l', D)$
    $\mathcal{S} \leftarrow$ PushToLeaf(Stash, $l$)              ▷ see § 4
    WritePath($l, \mathcal{S}$)
    return $D$
function ReadPath($l$)
    for $i \leftarrow 0$ to $L$ do                               ▷ read path
        bucket $\leftarrow \mathsf{Decrypt}_K(\mathcal{P}(l)[i])$
        InsertBlocks(Stash, bucket)
function WritePath($l, \mathcal{S}$)
    for $i \leftarrow 0$ to $L$ do                               ▷ write path back
        bucket $\leftarrow \mathcal{S}[i * L, \ldots, i * L + Z - 1]$
        RemoveBlocks(Stash, bucket)
        $\mathcal{P}(l)[i] \leftarrow \mathsf{Encrypt}_K$(bucket)
```

Table 2: Bucket fields. For reference, we illustrate field sizes for two extreme parameterizations: a 1 GB ORAM with 4 KB blocks and a 1 TB ORAM with 64 B blocks. We use $Z = 4$ as suggested by prior work [20, 30].

Name	Asymptotic Size	Practical size (in bits, $Z = 4$)
	Bucket header	
Encryption initialization vector	λ_t (§ 3.1.1)	64
Block address (U field)	$Z * O(L)$	72 to 136
Block leaf (L field)	$Z * O(L)$	“”
	Bucket data	
Block data (B field)	$Z * B$	131072 to 2048

block a is now in Stash and can be read/updated and remapped (Line 12-17). Finally, the backend tries to "push back"/re-encrypt as many blocks from Stash back to the ORAM tree as possible (Lines 26-29).

Implicit in Algorithm 1, each block is stored in the stash and ORAM tree alongside its program address and current leaf. Each bucket contains Z blocks, as well as their addresses/leaves and an initialization vector used for symmetric encryption. We call the latter information the *bucket header*. All bucket fields are shown in Table 2. We denote the address/leaf/initialization vector metadata as the bucket header. Dummy blocks are stored in the ORAM tree with a special program address $\perp$.

$\mathsf{PushToLeaf}(\mathsf{Stash}, l)$ on Line 18 yields an array of blocks in the order that they should be written back to path $\mathcal{P}(l)$ of the ORAM tree. $\mathcal{S}[i]$ represents the block to be written back to the i-th position on Path $\mathcal{P}(l)$, of which there are $(L + 1) * Z$. To keep the stash small, $\mathsf{PushToLeaf}(\mathsf{Stash}, l)$ needs to evict as many blocks as possible from Stash to path $\mathcal{P}(l)$. Performing this step efficiently is a big challenge for hardware designs [20] and we propose a simple mechanism in § 4 that solves the problem for any reasonable block size or memory bandwidth.

3.1.1 Path ORAM Security

The basic intuition for Path ORAM's security is that every PosMap lookup (Line 7) will yield a fresh random leaf that has never been used to access the ORAM tree before. This makes the sequence of ORAM tree paths accessed and the program address trace independent. Further, probabilistic encryption ensures that *which* block is read on the path is computationally indistinguishable. [30] proves that Stash capacity can be bounded to be small if $Z \geq 5$.

There are two security parameters, denoted λ and λ_t. We set $\lambda = 80$ and provision the stash capacity C to attain a stash failure probability of $2^{-\lambda}$. We set $\lambda_t = 64$ and assert that the ORAM cannot make more than 2^{λ_t} accesses. In practice, λ_t is the width given to several types of monotonically increasing counters; e.g., the initialization vectors used for symmetric encryption. Such counters must be wide enough so that no counter overflows.[1]

3.2 Recursive Path ORAM

In basic Path ORAM, the number of entries in the PosMap (§ 3.1) scales linearly with the number of data blocks in the ORAM. This results in a significant amount of on-chip storage. Recursive ORAM was first proposed by Shi et al. [27] to solve this problem and has been studied through simulation in trusted hardware proposals [8, 25]. The idea is to store the PosMap in a separate ORAM, and store the new ORAM's (smaller) PosMap on-chip.

We refer to the original ORAM as the *Data ORAM*, denoted as ORam_0, and call the second ORAM a *PosMap* ORAM, denoted ORam_1. Suppose each block in ORam_1 contains X leaf labels ($X = 16$ is typical

[1] If $\lambda_t = 64$ and each counter increments once per ORAM access, no counter will overflow even after 100s of years of constant accesses.

for $B = 512$), which correspond to X data blocks in ORam_0. Then, for a block with program address a_0 in ORam_0, its leaf label is stored in block $a_1 = a_0/X$ of ORam_1. Thus, every X consecutive blocks ($a_0, a_0+1, \ldots, a_0+X-1$ where a_0 is a multiple of X) share the same PosMap ORAM block. We note that this, and later, division operations are rounded down (floored) to the nearest integer. Further, hardware designs typically round X to the nearest power of 2 to simplify the division operation.

Accessing data block a_0 in the recursive construction involves two ORAM accesses. The first is to ORam_1 for block a_1, which contains the leaf label l_0 of block a_0 (this replaces Lines 7-8 in Algorithm 1). The second is to ORam_0 for block a_0.

Of course, the new on-chip PosMap might still be too large. In that case, additional PosMap ORAMs ($\mathsf{ORam}_2, \mathsf{ORam}_3, \ldots, \mathsf{ORam}_{H-1}$) may be added to further shrink the on-chip PosMap. The PosMap blocks a_i ($i > 0$) are analogous to page tables in conventional virtual memory systems, where the leaf labels are pointers to the next-level page tables or the pages. A recursive ORAM access is conceptually similar to a full page table walk.

The cost of recursive ORAM is longer latency: now we have to access all the ORAMs in the recursion on each ORAM access. In fact, not intuitively, with small block sizes and a large ORAM capacity, PosMap ORAMs can contribute to more than half of the total ORAM latency. We optimize the recursive ORAM construction in § 5 by utilizing the locality in PosMap ORAM accesses.

4 Stash Management

As mentioned in § 3.1, PushToLeaf() is a challenge for efficient Path ORAM hardware designs. Conceptually, PushToLeaf(Stash, l) needs to push every block in the Stash to the deepest possible bucket on path $\mathcal{P}(l)$ while maintaining the Path ORAM invariant. Phantom demonstrated that skipping this step (i.e., evicting blocks in "unsorted order" [20]) causes the Stash to grow uncontrollably.

On the other hand, Phantom describes how sorting the Stash (as described in [30]) can cause serious hardware performance bottlenecks. As they point out, a naïve implementation is to scan the Stash for each location on path $\mathcal{P}(l)$, which takes $O(C + L * Z)$ cycles per block. Similarly, Gentry et al. [9] suggest an $O(L)$ method. Phantom then proposes a hardware-optimized *heap sort* algorithm to perform Stash management, reducing this to $O(\log(C + L * Z))$ cycles per block. This is similar to the idea of Chung et al. [5], who suggest a binary search tree to perform "range queries" on Stash.

With Phantom's parameters, the pipelined heap sort design takes 11 cycles to evict a block (see Appendix A of [20]). Assuming a memory bandwidth between 512-1024 bits/cycle for modern FPGAs, this constrains the block size B to be $\geq$ 512-1024 bits times 11, to hide the heap-sort latency. In other words, when the block size is $<$ 704 to 1408 Bytes, the the ORAM controller's performance bottleneck is Stash management logic, rather than memory bandwidth.

4.1 PushToLeaf With Bit Hacks

We propose a new, simple PushToLeaf() implementation based on bit-level hardware tricks that takes a single cycle to evict a block. This *eliminates* the above performance overhead for any block size and memory bandwidth.

Our PushToLeaf() design is shown in Algorithm 2. Conceptually, PushToLeaf() is a hardware circuit that sequentially, for each block in Stash, pushes that block (PushBack()) as far towards to the leaf bucket along $\mathcal{P}(l)$ as possible using combinational logic.

Suppose l is the current leaf being accessed. We represent leaves as L-bit words which are read right-to-left: the i-th bit indicates whether the i-th bucket's child is the left child (0) or right child (1). On Line 3, we initialize the contents of $\mathcal{S}$ to $\perp$, where $\perp$ represents a dummy block. Occupied is an $L+1$ entry memory that records the number of real blocks that have been pushed back to each bucket so far.

We now explain the PushBack() routine in detail. Line 14 first concatenates 0 to both l and l' and XORs these vectors together. t_1 now represents in which levels the paths $P(l)$ and $P(l')$ diverge. Line 15 then clears all remaining bits *except* for the *right-most set bit*. t_2 is now called "one-hot" (meaning it contains

Algorithm 2 Bit operation-based Stash scan. 2C stands for two's complement arithmetic.

```
 1: Inputs: The current leaf $l$ being accessed
 2: function PushToLeaf(Stash, $l$)
 3:     $\mathcal{S} \leftarrow \{\perp$ for $i = 0, \ldots, (L+1) * Z - 1\}$
 4:     Occupied $\leftarrow \{0$ for $i = 0, \ldots, L\}$
 5:     for $i \leftarrow 0$ to $C + L * Z - 1$ do
 6:         $(a, l_i, D) \leftarrow$ Stash$[i]$                          ▷ Leaf assigned to $i$-th block
 7:         $level \leftarrow$ PushBack$(l, l_i)$
 8:         if $level > -1$ then
 9:             $offset \leftarrow level * Z +$ Occupied$[level]$
10:             $\mathcal{S}[offset] \leftarrow (a, l_i, D)$
11:             Occupied$[level] \leftarrow$ Occupied$[level] + 1$
12:     return $\mathcal{S}$
13: function PushBack$(l, l')$
14:     $t_1 \leftarrow (l \oplus l')||0$                                  ▷ Bitwise XOR
15:     $t_2 \leftarrow t_1$ & $-t_1$                                 ▷ Bitwise AND, 2C negation
16:     $t_3 \leftarrow t_2 - 1$                                      ▷ 2C subtraction
17:     full $\leftarrow \{($Occupied$[i] \overset{?}{=} Z)$ for $i = 0$ to $L\}$
18:     $t_4 \leftarrow t_3$ & ~full                                  ▷ Bitwise AND/negation
19:     $t_5 \leftarrow$ reverse$(t_4)$                                 ▷ Bitwise reverse
20:     $t_6 \leftarrow t_5$ & $-t_5$
21:     $t_7 \leftarrow$ reverse$(t_6)$
22:     if $t_7 \overset{?}{=} 0$ then
23:         return $-1$                                               ▷ Block is stuck in Stash
24:     return $\log_2(t_7)$                                          ▷ Note: $t_7$ must be one-hot
```

exactly 1 set bit) and its set bit indicates the *first* level where $P(l)$ and $P(l')$ diverge. Line 16 converts t_2 to a vector of the form $000\ldots111$, where set bits indicate which levels the block *can* be pushed back to. Line 18 further excludes buckets that already contain Z blocks (i.e., from previous calls to PushBack()). Finally, Lines 19-21 turns all current bits off except for the *left-most set bit*, which indicates the highest level towards the leaves that the block can be pushed back to.

Since our PushToLeaf() routine does not assume any order of blocks in Stash, we can also implement InsertBlocks() from Algorithm 1 in 1 cycle per block. In our current design, we manage Stash as a simple linked-list and simply add a node to the list to insert a block.

4.2 Hardware Implementation

Algorithm 2 runs $O(C + L * Z)$ iterations of PushBack() per ORAM access. $O(C)$ iterations are spent scanning blocks that may have been in the stash at the beginning of the access. The remaining iterations process blocks on the current path. In hardware, we pipeline Algorithm 2 in three respects to hide this $O(C + L * Z)$ operation.

First, the PushBack() circuit itself is pipelined to have (amortized) throughput of 1 block / cycle. PushBack() itself synthesizes to simple combinational logic where the most expensive operation is two's complement arithmetic of $(L+1)$-bit words. We note that reverse() costs no additional logic and the other bit operations (including $\log_2(x)$ when x is one-hot) synthesize to inexpensive circuits. To meet our FPGA's clock frequency, we had to add 2 pipeline stages after Lines 15 and 16. Thus, performing C iterations of PushBack() requires $C + 3$ cycles.

Second, as soon as an ORAM access starts and the leaf being read is presented to the backend (i.e., concurrent with Line 11 in Algorithm 1), blocks already in the stash are sent to the PushBack() circuit "in the background". Following the previous paragraph, suppose $C + 3$ is the number of cycles it takes to perform the background scan in the worst case. If the first block from DRAM arrives at the Stash in $\geq C+3$ cycles, the scan latency is hidden and we incur no performance loss. Generally, if the first block on the path arrives at the stash after C' cycles, the stash must stall for $\max(0, C + 3 - C')$ cycles before processing that block to make each ORAM access timing data-independent (§ 2). This wait latency is paid once per ORAM access.

In practice, hardware overheads (e.g., decryption latency, DRAM read latency, etc) cause $C+3-C'$ to be small. Using the methodology from [20], we extrapolate that $C = 78$ is sufficient for $\lambda = 80$ (§ 3.1.1). Given hardware implementation artifacts, C' is typically in the tens of cycles. We estimate C' for our prototype in § 8.2.

Third, after cycle $C + 3$, we send each block read on the path to the $\mathsf{PushBack}()$ circuit *as soon as it arrives from DRAM.* Since a new block can be processed each cycle, we are guaranteed to be able to start writing back blocks to DRAM 3 cycles after the *last block* read from DRAM is processed by $\mathsf{PushBack}()$.

5 Unified Path ORAM

As mentioned in § 3.2, the recursive ORAM construction is highly desirable for hardware designs due to the client storage reduction, but imposes significant performance penalties. In this section, we introduce *Unified Path ORAM*, or *Unified ORAM* for short, to handle recursion. Unified ORAM was first proposed in our previous work [23]. We describe the algorithm again, discuss its security under the definition of this paper, and present pseudocode in Appendix A.

The key observation of unified ORAM is that PosMap ORAM accesses have locality. For simplicity, suppose there is one PosMap ORAM named ORAM_1. Suppose the user program is linearly scanning memory; e.g., sending the ORAM controller the address trace $\{a, a+1, a+2, \cdots\}$.[2] For each of these accesses, the PosMap block needed is given by $\lfloor a/X \rfloor$, $\lfloor (a+1)/X \rfloor$, $\lfloor (a+2)/X \rfloor$, *etc.* The key point is that for $X > 1$, *the same PosMap block will be needed multiple times.*

This section develops a scheme to exploit this PosMap block locality. First in § 5.1, we introduce a new structure in the ORAM controller called the *PosMap Lookaside Buffer* (PLB for short), a novel mechanism for *caching PosMap blocks*. Second in § 5.2, we resolve a security problem for PLBs which completes the Unified ORAM scheme. To simplify the presentation, we present key ideas in this section, and precise pseudo-code is given in Appendix A.

5.1 PosMap Lookaside Buffer (PLB)

When we introduced recursive Path ORAM in § 3.2, we intentionally compared it to virtual memory and PosMap blocks to page tables. Conventional virtual memory systems have Translation Lookaside Buffers (TLBs) to cache page tables. Similarly, the PosMap Lookaside Buffer (PLB) aims at reducing the number of accesses to PosMap ORAMs by caching PosMap blocks on-chip.

As in § 3.2, we refer to the data ORAM as ORam_0, and to the PosMap ORAM hierarchy as $\mathsf{ORam}_1, \ldots, \mathsf{ORam}_{H-1}$. On an ORAM access to address a_0, recursive Path ORAM has to access H ORAMs in the recursion in decreasing order (starting with ORam_{H-1} first). For each PosMap ORAM ORam_i we add a PLB, referred to as PLB_i. As PosMap ORam_i is accessed, the requested PosMap block is added to PLB_i as if the PLB were a normal cache. If block $a_i = a_0/X^i$ is already in PLB_i before ORam_i is accessed, the ORAM controller directly starts the access from ORam_{i-1}, *skipping* ORam_i *and all the smaller PosMap ORAMs.*

Unfortunately, *just* adding PLBs as described above violates ORAM security. Now the adversary learns not only the total number of ORAM accesses (LLC misses), but also the *sequence* of ORAMs accessed in time (e.g., $\mathsf{ORam}_1, \mathsf{ORam}_0, \mathsf{ORam}_2, \mathsf{ORam}_1, \mathsf{ORam}_0, \ldots$). In information theoretic terms, this leakage grows linearly with time.

[2] This, and more generally striding memory (e.g., $a, a+i, a+2i$, etc), is quite common in real programs.

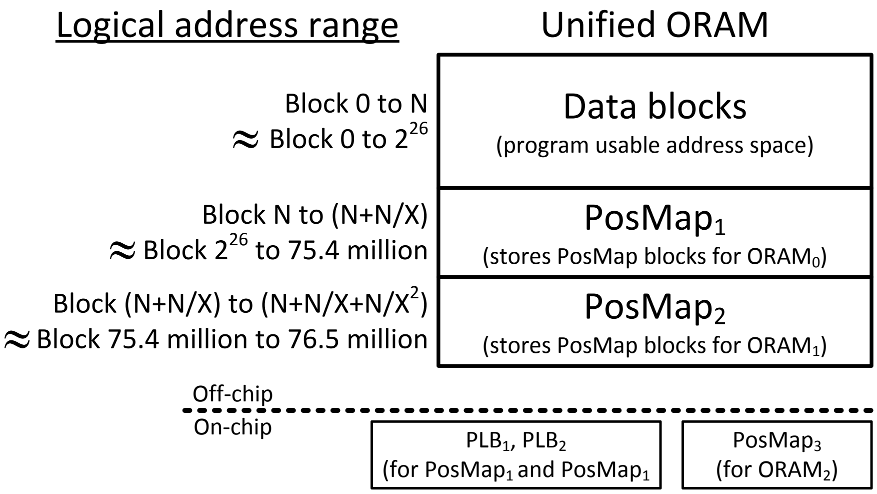
Logical address range
Unified ORAM
Block 0 to N
≈ Block 0 to 2^{26}
Data blocks
(program usable address space)
Block N to (N+N/X)
≈ Block 2^{26} to 75.4 million
PosMap$_1$
(stores PosMap blocks for ORAM$_0$)
Block (N+N/X) to (N+N/X+N/X^2)
≈ Block 75.4 million to 76.5 million
PosMap$_2$
(stores PosMap blocks for ORAM$_1$)
Off-chip
On-chip
PLB$_1$, PLB$_2$
(for PosMap$_1$ and PosMap$_1$
PosMap$_3$
(for ORAM$_2$)

Figure 1: Unified ORAM address space assuming there are $N = 2^{26}$ data blocks and $X = 8$.

5.2 Unified ORAM

Our key idea to fix the PLB insecurity is to store the Data ORAM and all PosMap ORAMs in the same *Unified* ORAM tree, thereby making the ORAM access sequence indistinguishable from any other access sequence of the same length. In a Unified ORAM, we still need a hierarchical PosMap $\{\mathsf{PosMap}_1, \mathsf{PosMap}_2, \cdots, \mathsf{PosMap}_{\mathsf{H}-1}\}$, where PosMap_1 denotes the PosMap for the data blocks, and PosMap_{h+1} denotes the PosMap for PosMap_h ($h \geq 1$) with the property $|\mathsf{PosMap}_{h+1}| = |\mathsf{PosMap}_h|/X$. Crucially, there is only one Path ORAM tree that contains all the data blocks *and* the PosMap blocks (for PosMap_i where $1 \leq i \leq H$). Note that the data blocks and PosMap blocks are of the same size.

Different blocks occupy different logical address spaces in the Unified ORAM, as illustrated in Figure 1. If there are N data blocks, they will occupy address space $[0, N)$. This is the memory space seen by the programs. Addresses beyond N are reserved for PosMaps and are not accessible to the user program. Each PosMap is X times smaller than the previous one, so the PosMap block storage overhead is small. As shown in Figure 1, PosMap_1 occupies address $[N, N + N/X)$, PosMap_2 occupies address $[N + N/X, N + N/X + N/X^2)$, and so on. For a data block with address a_0 ($a_0 < N$), its first-level PosMap block is the (a_0/X)-th block in PosMap_1, which has address $a_1 = N + a_0/X$; its second-level PosMap block is the (a_0/X^2)-th block in PosMap_2, which has address $a_2 = N + N/X + a_0/X^2$, and so on. As before, the smallest PosMap (PosMap_3 in Figure 1) is stored on-chip.

Our final implementation manages a single PLB whose space is shared by all PosMaps (§ 8). In this case, the steps to access a data block with address a_0 ($a_0 < N$) in unified ORAM are shown below.

1. **(PLB lookup)** For $h = 0, \ldots, H - 1$, look up the PLB for the leaf label of block a_i. If hit, go to Step 2; else, continue (a_{H-1} will definitely hit since its leaf label is in the on-chip PosMap).
2. **(PosMap block accesses)** For $i = h, \ldots, 1$, Access the unified ORAM for block a_i and put it into the PLB. If this evicts another PosMap block from the PLB, add that block to the stash. (This loop will not be entered if $h = 0$.)
3. **(Data block access)** Access the unified ORAM for block a_0 and return it to the last-level cache.

In Step 2 and 3, the leaf label for block a_h is originally in PLB, or in the on-chip PosMap if $h = H - 1$. The leaf labels for the other blocks a_i ($0 \leq i < h$) are obtained from the unified ORAM tree, and blocks a_j ($1 \leq j < h$) are brought into the PLB.

PLB Hardware Implementation. The PLB(s) may be architected as any type of cache such as a set-associative cache. We assume a direct-mapped cache for the rest of the paper mainly for its design simplicity. In whichever form, the PLB adds hardware area to the design in the form of on-chip storage. That said, it is unclear whether Unified or baseline Recursive ORAM requires more storage (e.g., Unified ORAM only requires one stash, as opposed to one stash per ORAM).

5.3 Security

We now give a proof sketch that Unified ORAM achieves the security definition in § 2.

Proof. All Unified ORAM (frontend) requests to main memory go through the ORAM backend. We remark that a Path ORAM backend (denoted $\mathsf{PORAMBackend}(a, l, l', op, D')$) attains the security definition in § 2 if, for all calls to the backend, the input leaf l is always random and fresh. Unified ORAM calls the backend in two cases: First, if there is a PLB hit and the backend request is for a PosMap or Data block. In that case, the leaf l sent to the backend is stored in a PosMap block that was in the PLB, and by the Recursive ORAM algorithm was remapped to l at the instant the block was last accessed. Second, if there is a PLB miss and the backend request is for a PosMap block. In that case, the leaf l comes from the on-chip PosMap which also guarantees its freshness. □

PLB hit and miss rates will affect the ORAM sequence length $|\mathsf{ORAM}(\overleftarrow{S})|$ by, conceptually, filtering out some accesses. Now $|\mathsf{ORAM}(\overleftarrow{S})|$ is determined by, and thus reveals, the sum of LLC misses and PLB misses. Since both processor cache and the PLB are on-chip, adding the PLB is (security-wise) equivalent to adding another level of processor cache: in both cases, the adversary's view is the total number of ORAM accesses made over the course of the program's execution.

5.4 Generality of Construction

We remark that Unified ORAM impacts $\mathsf{PORAMFrontend}()$ from Algorithm 1 only. Besides Path ORAM, it works with other ORAM constructions that require a PosMap, including [27, 29, 9], and also *RAW Path ORAM*, which we present in a later section. These constructions all have large client-side storage, and all need recursion to be used in a trusted hardware setting.

6 RAW Path ORAM

We now discuss a second extension to Path ORAM which we call $\mathrm{R}^{A+1}\mathrm{W}$ Path ORAM, or *RAW ORAM* for short. RAW ORAM reduces our design's area footprint and dramatically simplifies the proof of security for Path ORAM. The RAW algorithm impacts $\mathsf{PORAMBackend}()$ from Algorithm 1 only; it can be used with or without Unified ORAM. As with Unified ORAM, we present key ideas in this section. Precise pseudo-code and the complete proof of security is given in Appendix B and C, respectively.

6.1 Overview

Parameter A. We introduce a new parameter A, set at system boot time. RAW Path ORAM splits $\mathsf{PORAMBackend}()$ into two flavors: *read-only* (RO) and *read-write* (RW) accesses. For a given A, RAW Path ORAM obeys a strict schedule that the ORAM controller performs one RW access after every A RO accesses.

An **RO access** performs *only* the operations needed to read the requested block into Stash, and logically remove it from the ORAM tree. The requested block is then returned or updated as with basic Path ORAM. This corresponds to Lines 11-17 in Algorithm 1 with three important changes. First, we will only decrypt the minimum amount of information needed to *find* the requested block and add it to Stash. Precisely, we decrypt the Z block addresses stored in each bucket header (§ 3.1), to identify the requested block, and then decrypt the requested block itself (if it is found). The amount of data read vs. decrypted is illustrated in Figure 2.

Second, we add only the requested block to the Stash (as opposed to the whole path). Third, we update the bucket header containing the requested block to indicate a block was removed (e.g., by changing its program address to $\perp$), and re-encrypt/write back to memory the corresponding state for each bucket. To re-encrypt header state only, we encrypt that state with a second initialization vector denoted IV_2. The rest

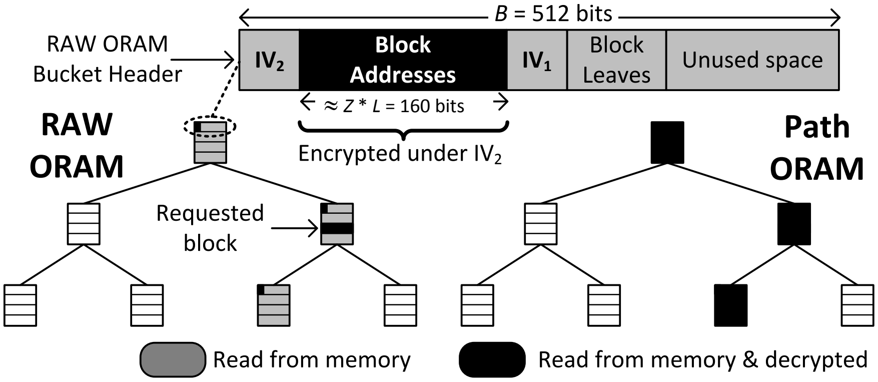

Figure 2: Data read vs. data decrypted on a RAW ORAM RO read (left) and Path ORAM access (right) with $Z = 3$. IV_1 and IV_2 are initialization vectors used for encryption.

of the bucket is encrypted under IV_1. A strawman design may store both IV_1 and IV_2 in the bucket header (as in Figure 2). We detail an optimized design in § 6.4.

An **RW access** performs a normal (read+writeback) but *dummy* ORAM access to a static sequence of leaves (described in § 6.2). Dummy accesses skip Lines 12-17 in Algorithm 1—i.e., their only purpose is to evict blocks from Stash. RW accesses occur over a static ordering of paths corresponding to *a reverse lexicographic ordering*, which will be discussed in detail in § 6.2.

Memory Bandwidth. We compare the bandwidth needed to serve A frontend requests for RAW ORAM and Path ORAM. For both proposals, we assume that bucket headers are stored externally, alongside each bucket and padded to the data block size (64 Bytes). Thus, there are $(L + 1) * (Z + 1)$ blocks on a path. The RAW ORAM header update operation writes $(L + 1)$ blocks back to the ORAM tree. RAW ORAM bandwidth relative to Path ORAM bandwidth is then given by $\frac{A+(A+2)*(Z+1)}{2*A*(Z_p+1)}$, where Z_p is a competitive Z value for Path ORAM and was suggested to be 4 by prior work [20, 30].

Encryption Bandwidth. On an RO access, we must decrypt and then re-encrypt $Z * L$ bits per bucket (amortized), compared to $Z_p*(2*L+B)$ for normal Path ORAM. With $B = 512$, RO accesses require $> 10\times$ less encryptions than Path ORAM; thus we will not include it in further analysis. To service A frontend requests, RAW ORAM must then perform $\frac{Z+1}{A*(Z_p+1)}$ encryption operations relative to Path ORAM.

Parameter recommendations. We experimentally determine A and Z that give negligible Stash overflow probability as well as good memory and encryption bandwidth. Experimental results in Figure 3 suggest that with $Z = 5, A = 5$ (Z5A5) or $Z = 4, A = 3$ (Z4A3), Stash overflow probability decreases exponentially with Stash size.

Plugging in parameters from the above analysis, Z5A5 achieves 6% memory bandwidth improvement and $\sim 4\times$ encryption reduction over Path ORAM. Z4A3 achieves 7% memory bandwidth improvement and $\sim 3\times$ encryption reduction. We will use Z5A5 in the evaluation and remark that extrapolated to $\lambda = 80$, this configuration requires $C = 64$. In Appendix C, we formally prove that several less competitive configurations have negligible Stash overflow probability.

6.2 Path Write Predictability

Gentry et al. [9] proposed writing paths in *reverse lexicographic order* to simplify the analysis on Stash overflow probability. We make two key observations relating to reverse lexicographic order in this work. First, reverse lexicographic order improves eviction quality by load-balancing paths. This enables $Z5A5$ without causing Stash overflows. More important, it gives RAW ORAM a property that we call *path write predictability*, *PWP* for short, which we now explain in detail.

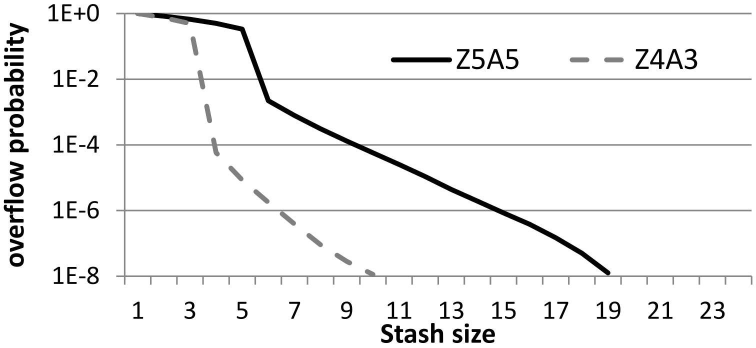

Figure 3: Stash size corresponds to parameter C from Table 1. For RAW ORAM with $A = 5, Z = 5$ and $A = 4, Z = 3$, experiments show that Stash overflow probability drops exponentially with Stash size. We experiment with $L = 25$, simulate for over 1 billion RO accesses, and sample the stash occupancy *before* each RW access.

Let G be the number of RW accesses made so far (where $|G| = \lambda_t$; see § 3.1.1). At startup, the ORAM controller sets $G = 0$ and increments G once after each RW access. The ORAM leaf that is accessed on each RW access is then simply the low-order L bits in G, namely $G \mod 2^L$.

Key observation: Given G, we can determine exactly how many times any bucket along any path has been *written* in the past. Specifically, due to load-balancing nature of reverse lexicographic order, if $\mathcal{P}(l)[i]$ has been written g_i times in the past, then $\mathcal{P}(l)[i + 1]$ has been written $g_{i+1} = \lfloor (g_i + 1 - l_i)/2 \rfloor$ where l_i is the i-th bit in leaf l.[3]

6.3 Security

RO accesses always read paths in the ORAM tree at random, just like Path ORAM. RW accesses occur at predetermined time (always after A RO accesses) and are to predictable/data-independent paths. It remains to analyze the Stash occupancy and prove that Stash overflow probability is negligible in Stash size, which we prove in Appendix C. Interestingly, the deterministic write pattern (PWP) simplifies the proof dramatically when compared to the original Path ORAM proposal [30].

6.4 AES Area Savings In Practice

Despite RAW ORAM's theoretic area savings for encryption units, careful engineering is needed to prevent that savings from turning into performance loss. The basic problem is shown in Figure 4.

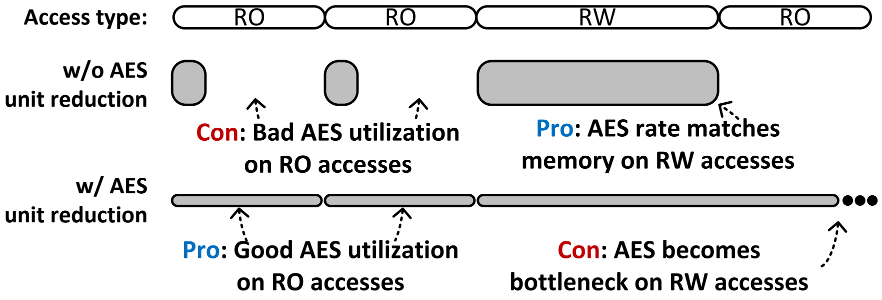
Access type:
RO
RO
RW
RO
w/o AES unit reduction
Con: Bad AES utilization on RO accesses
Pro: AES rate matches memory on RW accesses
w/ AES unit reduction
Pro: Good AES utilization on RO accesses
Con: AES becomes bottleneck on RW accesses

Figure 4: Potential RAW ORAM performance bottleneck. AES refers to symmetric encryption.

[3] This can be easily computed in hardware as $g_{i+1} = (g_i + \sim l_i) \gg 1$, where $\gg$ is a right bit-shift and $\sim$ is bit-wise negation.

In Figure 4, *w/o AES unit reduction* refers to a RAW ORAM built with the same number of encryption (AES) units as Path ORAM. In this design, there are enough AES units to rate match memory on a RW access; thus RW accesses take the same amount of time as Path ORAM accesses. But on RO accesses, AES units are left idle most of the time because RO accesses require less encryptions.

Of course, we wish to reduce the number of AES units by $\sim 4\times$ as suggested in § 6.1. If we do this (Figure 4, *w AES unit reduction*) on RO accesses, the reduced number of AES units rate match memory. On RW accesses, however, DRAM transfers data faster than the available AES bandwidth, creating a performance bottleneck. Specifically, each RW access will take roughly $\sim 4\times$ longer to complete, proportional to the reduction in AES units.

All symmetric encryption/decryption for the rest of the paper is assumed to be given through AES-128 in CTR mode. The key idea to elegantly hide AES latency for the reduced area design comes from path write predictability (§ 6.2): since we know which paths will be read/written on future RW accesses, we can *pre-compute* the AES-CTR initialization vector IV_1 (§ 6.1). In other words, we can generate RW AES masks "in the background" during concurrent RO accesses.

To decrypt the i-th 128-bit ciphertext chunk of the bucket with unique ID $BucketID$, as done on an RW ORAM path read, we XOR it with the following mask:

$$\mathsf{AES}_K(g \mid\mid BucketID \mid\mid i)$$

where g is the bucket write count (§ 6.2). Correspondingly, re-encryption of that chunk on the RW path writeback is done by generating a new mask where the write count has been incremented by 1. We note that with this scheme, g takes the place of IV_1 and since g can be derived internally, we need not store it externally.

On both RO and RW accesses, we must decrypt the remaining bucket header data (i.e., each block's program address and block valid bits from § 6.1). For this we apply the same type of mask as in Ren et al. [25], namely $\mathsf{AES}_K(\text{IV}_2 \mid\mid BucketID \mid\mid i)$, where IV_2 is stored externally as part of each bucket's header.

At the implementation level, we time-multiplex an AES core between generating masks for IV_1 and IV_2. The AES core prioritizes IV_2 operations; when the core is not servicing IV_2 requests, it generates masks for IV_1 in the background.

7 Integrity Verification

We now describe a novel and simple integrity verification scheme that achieves asymptotic improvements in hash bandwidth and removes a serialization bottleneck, relative to prior proposals.

Our key observation is that the PosMap can double as a table that tracks each block's *access count*. Our scheme replaces each leaf in the PosMap with a monotonically increasing counter. When a block is accessed, we lookup the PosMap to get the block-of-interest's counter. The counter is first converted into a random leaf by means of a pseudorandom function (PRF), which implements the functionality of the original PosMap (§ 3.1). To remap a block, we simply increment the counter for the block. Once the block is fetched from ORAM, we produce/check a MAC for the requested block using the block's counter, address and data. This MAC verifies the authenticity/freshness of that block. The hard problem this scheme solves is replay attacks: the adversary cannot replay a stale data block because the *monotonically increasing and un-tamperable* counter is used as an input to the MAC [26]. Crucially, the scheme only needs to integrity check *a single block* per ORAM access, as opposed to all blocks on the ORAM tree path [24].

7.1 Detailed Description

We now describe the changes made to $\mathsf{AccessORAM}()$ from Algorithm 1 — i.e., for position-based ORAM schemes without Recursion — to support our scheme:

1. **PosMap format and lookup.** The PosMap now stores, for each block, a monotonically increasing counter which represents the number of times that block has been accessed. To lookup the PosMap

for a block with address a (as in $\mathsf{PORAMFrontend}()$ from Algorithm 1), we perform the following operations:

$$\begin{aligned} c &\leftarrow \mathsf{PosMap}[a] && \text{Read access counter} \\ c' &\leftarrow c+1 && \text{Increment counter (needed to generate fresh leaves and MACs)} \\ \mathsf{PosMap}[a] &\leftarrow c' && \text{Update PosMap with new leaf/counter} \\ l &\leftarrow \mathsf{PRF}(a||c) && \text{Create block's current leaf} \\ l' &\leftarrow \mathsf{PRF}(a||c') && \text{Create block's remapped leaf} \end{aligned}$$

l and l' are leaves and serve the same purpose as in the original protocol. c and c' are counters and will be used in later steps for integrity checking. We remark that $\mathsf{PRF}()$ may be implemented using a block cipher such as $\mathsf{AES}_K()$, since each input to the PRF (an address and monotonic counter) is unique.

2. **Per-block MACs.** Suppose a data block has address a, data d and current leaf l in the original protocol (§ 3.1). That is, the tuple (a, l, d) is stored in the ORAM tree and Stash. Our scheme extends the block to include the following MAC:

$$h = \mathsf{MAC}_K(c \mid\mid a \mid\mid d)$$

where c is the current access count for that block and $l = \mathsf{PRF}(a||c)$. The block $D = (h, a, l, d)$ is the new unit of information stored in the tree/Stash and behaves exactly as did blocks in the original protocol.

3. **Checking integrity/freshness.** Suppose the ORAM frontend sends the backend a request for some block with address a, current leaf l and remapped leaf l'. As before, the access count c for the block satisfies $l = \mathsf{PRF}(a||c)$. Suppose the backend returns the tuple $(h\star, a\star, d\star)$, where $\star$ indicates the adversary may have tampered with that value. Immediately before the frontend receives the block (i.e., after Line 13 in Algorithm 1), we add the following assertion to check the *requested block's* authenticity/freshness:

$$\text{assert } h\star == \mathsf{MAC}_K(c \mid\mid a \mid\mid d\star)$$

Note that d is read from memory (and thus can be tampered with) and K,c,a are produced internally (and thus cannot be tampered with). Failing this assertion constitutes an integrity violation. After Line 17, we update the MAC for the block. Suppose the ORAM operation is a write that updates $d\star$ to d'. Then the new MAC is given by:

$$h' = \mathsf{MAC}_K(c' \mid\mid a \mid\mid d')$$

and $D' = (h', l', a, d')$ are written back to the ORAM tree.

Extending the scheme to Recursive position-based ORAMs. To achieve a small/constant-sized on-chip PosMap, we can re-apply the recursive ORAM technique (§ 3.2). In this case, the leaves in each PosMap block are replaced by per-block counters and these counters are used to generate the current leaf and MAC for each block in the next level PosMap. We discuss overheads and optimizations for this scheme in § 7.4.1.

7.2 Key Advantage: Hash Bandwidth and Parallelism

To perform a non-recursive ORAM access (i.e., read/write a single path), Path ORAM reads/writes $O(\log N)$ blocks from external memory. Prior Merkle tree constructions [24, 2] must integrity verify $O(\log N)$ blocks in order to check/update against a trusted root hash. Our construction must integrity verify (check and update) $O(1)$ blocks — namely the block of interest — per access.

To give a sense for overheads, we consider two extreme scenarios for the Merkle tree design. For all examples, we assume $Z = 4$ and remark that there are $(L + 1) * Z$ blocks on an ORAM path. To minimize the Merkle tree's disadvantage, we consider a small ORAM (1 GB) that uses large blocks (4 KB). In that case, $L = 18$ and 76 blocks are integrity checked per access. On the opposite extreme, we consider a 1 TB ORAM with 64 B blocks. In that case, $L = 34$ and 140 blocks are integrity checked. Some additional data that we don't account for — such as sibling hashes — is also integrity checked in the Merkle tree scheme. With our new integrity scheme, for both configurations, only a single block is integrity checked.

Another important point is that integrity verifying only a single block prevents a serialization bottleneck present in Merkle tree schemes. Consider the scheme from [24] which assumes Path ORAM. On a path writeback, each Merkle tree node's hash must be recomputed based on the new data in the corresponding ORAM tree bucket *and that Merkle tree node's child hashes.* Thus, updating the hash tree is a fundamentally sequential operation and if this process cannot be hidden by the path writeback operation, it will be the system's performance bottleneck.

7.3 Subtly Broken Schemes

As we just described, our scheme only integrity verifies the block of interest. It *does not* integrity verify the per-bucket initialization vector (IV) used for encryption (§ 3.1). However, not integrity checking the IV and using the IV generation scheme from [25] causes security to break.

In [25], as we also assume in Table 2, each bucket contains the IV (referred to as $BucketCounter$ in that work) for that bucket. Suppose some bucket is read on an access whose IV is currently $BucketCounter$. Then the IV written back to that bucket on the path writeback is $BucketCounter + 1$. The OTP used to encrypt that bucket is $\mathsf{AES}_K(BucketID||BucketCounter)$ where $BucketID$ is a unique per-bucket identifier.

The attack, using this scheme, proceeds in two phases. In phase A, the program running on the processor writes some values to ORAM that are known to the adversary. Call this data D_{public}. In phase B, the program runs on secret data. Call this data D_{secret}. For example, the user's program may contain a trojan that gets some CPU time before the user's actual program. The attack works as follows:

1. During phase A, the adversary records the contents of some bucket with bucket ID $BucketID$ and IV $BucketCounter$. This ciphertext is $\mathsf{AES}_K(BucketID||BucketCounter) \oplus D_{public}$. Since the adversary knows D_{public}, it learns $\mathsf{AES}_K(BucketID||BucketCounter)$.
2. At some later time, during phase B, the adversary tampers with bucket $BucketID$ by replacing its IV with $BucketCounter - 1$ (i.e., an IV related to that from Step 1). When bucket $BucketID$ is read by ORAM, the contents of the bucket will decrypt to garbage. Yet, no integrity violation will be detected unless bucket $BucketID$ contained the block of interest. If no violation is detected, the bucket will get filled with new data (part of D_{secret}), and be written back to the ORAM as the ciphertext $\mathsf{AES}_K(BucketID||BucketCounter) \oplus D_{secret}$.
3. Since the adversary recorded $\mathsf{AES}_K(BucketID||BucketCounter)$, it can now decrypt this ciphertext to reveal D_{secret}.

The fix for this problem is relatively simple: To encrypt any bucket about to be written to DRAM, we now use the pad $\mathsf{AES}_K(GlobalCounter)$, where $GlobalCounter$ is now a single monotonically increasing counter stored in the ORAM controller ($|GlobalCounter| = \lambda_t = 64$ bits). When the bucket is fully encrypted, $GlobalCounter$ is written out alongside the bucket as its IV and the $GlobalCounter$ counter (stored in the ORAM controller) is incremented by J, where $J = \lceil Z * (2 * \log N + B) \rceil / B_{cipher}$ and B_{cipher} refers to the cipher block size (i.e., $B_{cipher} = 128$ for AES-128). Conceptually, J is the number of block cipher blocks per bucket. Thus, every bucket will always be encrypted with a fresh OTP which cannot be replayed (since it is seeded directly from the trusted on-chip counter $GlobalCounter$).

We remark that RAW Path ORAM (§ 6.4) can apply this scheme to prevent the above attack on IV_2. Since IV_1 in the RAW Path ORAM scheme is not stored externally, it does not need to be protected.

7.4 Other Overheads

7.4.1 PosMap Storage

Our integrity scheme increases the on-chip PosMap size (if non-recursive ORAM is used) or increases the number of levels of recursion needed for recursive ORAM.

Each entry in the original PosMap was approximately $\log N$ bits. However, each access counter in our scheme must be wide enough to never overflow. As in § 3.1.1, we assume each counter is $\lambda_t = 64$ bits wide to be conservative. Comparing the empirical values for B and $\log N$ in § 7.2 to λ_t, our integrity scheme increases the PosMap's size by a factor of $1.9 - 3.5\times$.

Let X now denote the number of counters per PosMap block. We note that to achieve the original $O(\log^2 N/\log X)$ asymptotic bandwidth result for recursive Path ORAM [30], it suffices to consider block sizes where $B = X\lambda_t$ for $X \geq 2$.[4] This holds for all $B \geq 128$ bits which shows that our technique applies for all popular block sizes (e.g., 64 B (this work), 128 B [25] and 4 KB [20]).

The number of levels of recursion is $O(\log N/\log X)$. Since $\log N < \lambda_t$ for practical configurations (§ 7.2), our integrity scheme causes the number of levels of recursion to increase. For example, to achieve a final PosMap size of $\approx 100\ KB$ with a 128 GB ORAM and $B = 512$, we have $X = 16$ with the original recursive scheme (§ 3.2), resulting in 5 levels of recursion. Our integrity scheme requires $X = B/\lambda_t = 8$, and 7 levels of recursion.

We observe that the compressed PosMap scheme from [23] can be applied to our construction to eliminate the additional levels of recursion. In that work, each leaf in each PosMap block is represented as an individual counter (IC) appended to a global counter (GC) where the global counter is shared among all leaves in the block. Each leaf is generated via $\mathsf{PRF}(a||GC||IC)$ where a is the block address. This construction is compatible with our notion of per-block access counts: for each block, that block's $GC||IC$ is never repeated. Thus, it can be applied to improve our integrity scheme's efficiency. [23] report that through compression, $X = 32$ can be achieved when $B = 512$ which outperforms baseline Recursive ORAM without our integrity scheme (§ 3.2).

7.4.2 MAC and Initialization Vector Storage

Bits stored externally. Each data block stored in the ORAM tree and Stash now consists of a MAC appended to the original block. Thus, each bucket (Table 2) stores $Z * \lambda_t$ additional bits. Simply comparing B and λ_t, our scheme incurs $\leq 12\%$, 6% and .2% space overhead for the popular block sizes 64 B, 128 B, and 4 KB respectively. We remark that the per-bucket initialization vector used for encryption/decryption ($GlobalCounter$ from § 7.3) is the same width as in the baseline Path ORAM scheme (§ 3.1).

Memory bandwidth. Practical performance depends on whether the additional MACs in each bucket cause the number of DRAM bursts per bucket to increase. In DDR3 DRAM, each burst is 512 bits. Then, assuming MACs are stored sequentially in external memory, our scheme adds at most one burst per bucket for $Z \leq 4$ and $\lambda \leq 128$. We note that when $Z = 4$ and $B = 512$ and 1024, each bucket requires 5 and 9 bursts, respectively, without including MACs. When $Z = 3$, any ORAM with ≤ 34 levels (this results in a 128 GB ORAM when $B = 512$) requires no additional bursts.

7.5 Security

We now give proof sketches for our scheme's integrity and privacy guarantees. We first give an argument for integrity since it is required for the proof of privacy to hold.

[4] We consider the case from [30] when block size for recursive/data ORAMs is uniform. This matches the unified ORAM scheme from § 5.

7.5.1 Integrity

Breaking our integrity verification scheme is as hard as breaking the underlying MAC, and thus attains the integrity definition from § 2 with overwhelming probability. We start with a simple lemma of our MAC scheme:

Lemma 1: Suppose some ORAM access is to a block with address a which the frontend associates with the counter c and that the backend returns the tuple $(h\star, d\star)$ for this access. If the counter c has not been used to create a MAC for block a in the past, producing an un-authentic/un-fresh forgery $(h\star, d\star)$ is as difficult as breaking the underlying MAC scheme.

Proof. Note that a and c come from the frontend. Thus the adversary cannot tamper with those values. To come up with a forgery, the adversary has to find a pair $(h\star, d\star)$ such that $h\star = \mathsf{MAC}_K(c||a||d\star)$. Since c is non-repeating, this construction immediately reduces to a standard MAC. □

We also have the following observation:

Observation 1: If all address and counter pairs (a_i, c_i) the frontend reads from the PosMap for the first k accesses have not been tampered with, then the frontend generates a unique (a_{k+1}, c_{k+1}) (i.e., $(a_i, c_i) \neq (a_{k+1}, c_{k+1})$ for $0 \leq i \leq k$). Moreover, $(a_i, c_i) \neq (a_j, c_j)$ for all $i \neq j$.

This property can be seen directly from the algorithm description. For every a, we have a unique counter that increments on each accesses. Since we have a monotonically increasing counter for each a, each address and counter pair will be different from previous ones.

We now use Lemma 1 and Observation 1 to prove that, for each ORAM access, the integrity verifier will only fail to detect the tampering of the block of interest if the underlying MAC scheme is broken. The intuition is that the counters in the on-chip PosMap form the root of trust and ensure that the first level PosMap blocks are fresh/authentic. The first level PosMap blocks form a root of trust for the second level blocks, and so on.

Theorem 1: Breaking the integrity scheme described in § 7.1 is as hard as breaking the underlying MAC scheme.

Proof. We proceed via induction on the number of accesses. In the first ORAM access, the frontend has not seen any (a, c) pairs. Thus (a_i, c_i)'s given to the frontend (the empty-set) have not been tampered by the adversary, and (a, c) generated by the frontend is unique. Therefore, the adversary cannot produce a valid MAC for block a (Lemma 1). Suppose the integrity verifier has detected no violations up to access $n-1$. If the adversary has not broken the underlying MAC, then the frontend sees (a_i, c_i)'s for $1 \leq i \leq n-1$ which have not been tampered with by the adversary. Therefore, it generates a unique (a_n, c_n) pair for the nth access (Observation 1). Then by Lemma 1, the adversary cannot generate a valid MAC for any block accessed up to that point (and, in particular, the nth access). Thus the adversary fails to generate a valid MAC with overwhelming probability, and breaking the integrity scheme is as hard as breaking the underlying MAC scheme. □

7.5.2 Privacy

Suppose access pattern $\overleftarrow{S}$ corresponds to the correct execution of some program (i.e., without an adversary performing any tampering). We show that our scheme leaks no more than 1 bit on top of the leakage intrinsic to revealing $|\mathsf{ORAM}(\overleftarrow{S})|$. We remark that the following analysis applies for any ORAM frontend (e.g., non-Recursive, Recursive or Unified). First, to simplify the analysis, we make the following change to our protocol:

Remark 1: *If, after starting an ORAM access, the outcome of the ORAM access results in anything except a successful integrity check for the block of interest, the ORAM backend (a) stops servicing requests from the frontend and (b) goes into an infinite loop and makes accesses to random paths forever (or until it is forcibly terminated by some managing entity). In that case, we say* $|\mathsf{ORAM}(\overleftarrow{S})| = \infty$ *and we assume that, from the adversary's point of view, this looks indistinguishable from the case where the program using* *the ORAM has gone into an infinite loop.* Generally, how integrity violations are handled when they are detected is a system-level concern and we remark that this strategy is not the only option.

We now show that at any point in a program's execution, the ORAM trace visible to the adversary leaks only its length.

Lemma 2: The observed ORAM trace at any point of the execution is computationally indistinguishable from any other ORAM trace of the same length.

Proof. We proceed via induction. At the time of the first ORAM access, we assume each bucket in the ORAM tree has been initialized in the following way: all block addresses per bucket are set to $\perp$ (indicating dummy blocks) and encrypted using the IV scheme from § 7.3. Thus, the ORAM tree is initially empty. For the first ORAM access, suppose we access path $\mathcal{P}(l)$. l is indistinguishable from any other random leaf, since $l = PRF(a||c)$ for a unique (a, c) (Observation 1). Furthermore, the data written back to $\mathcal{P}(l)$ looks like random bits, because all buckets written to DRAM are masked by fresh OTPs (§ 7.3). Thus, the first ORAM access is indistinguishable from any other single ORAM access.

Assume that Lemma 2 is true for the first $n-1$ accesses. We now analyze three cases. First, the program will not make another ORAM access. This case simply reduces to the case with $n-1$ accesses, and thus Lemma 2 holds. Second, the ORAM controller will make another access and has, for all prior accesses, checked each block returned to the ORAM frontend as fresh/authentic. Due to Theorem 1, this means the frontend has seen unique (a_i, c_i)'s for $1 \leq i \leq n-1$, unless the adversary has broken the underlying MAC scheme. Thus, the frontend will produce the a unique (a', c') pair (Observation 1), which implies $l' = PRF(a', c')$ sent from the frontend to the backend is indistinguishable from any other random leaf. Third, the ORAM controller has detected an integrity violation or some other abnormal behavior during some prior access. In that case, due to our scheme from Remark 1, the next leaf l' exposed by the backend has been generated randomly at that instant, and is indistinguishable from any other leafs. Finally, we note that in the later two cases, the bits written to DRAM for path $\mathcal{P}(l')$ looks random (§ 7.3). Thus, in the second or third case, the nth ORAM access is indistinguishable from any other ORAM access. Therefore, any ORAM trace generated by our scheme is computationally indistinguishable from all other ORAM traces of the same length. □

Given that leakage reduces to length of the ORAM trace, we now bound the additional leakage of our scheme over the termination channel to 1 bit.

Theorem 2: Under the scheme in Remark 1, the generated ORAM trace leaks at most 1 bit in addition to the length of the trace.

Proof. Since the adversary will observe the program's termination time anyway, we assume the adversary knows the value of $|\mathsf{ORAM}(S)|$, denoted T, a priori. Recall that $|\mathsf{ORAM}(S)|$ is the expected length of the program's ORAM trace without any tampering. So, we will assume a strictly more powerful adversary than might be the case in real life, as $|\mathsf{ORAM}(S)|$ may not be known a priori, for all programs.

Then, at time t, the adversary has access to the first t ORAM accesses $\mathsf{ORAM}(S)_t$, the termination time T, and its own tampering pattern. We have shown previously that in our scheme, the generated ORAM trace is computationally indistinguishable from any other ORAM trace of the same length. Thus the only information leaked by $\mathsf{ORAM}(S)_t$ is $|\mathsf{ORAM}(S)_t| = t$. Now consider the information leakage during 3 different time regimes: (1) $t < T$, (2) $t = T$, and (3) $t > T$.

In case (1), whether tampering happened or not, we do not terminate. Thus

$$Pr[\text{we terminate at } t | t < T, \mathsf{ORAM}(S)_t, T, \text{tamper}] = 0,$$

meaning the entropy is 0.

In case (2), we have

$$Pr[\text{we terminate at } T | \mathsf{ORAM}(S)_T, T, \text{tamper}] = p$$

for some p. For all values of p, the entropy is upper-bounded by 1.

Table 3: Design variants. See Table 1 for variable defs.

Configuration	Clock (Core/AES)	Z	A	PLB
recursive_basic_e3	200/300	4	N/A	N/A
unified_basic_e3	200/300	4	N/A	8 KB
unified_raw_e1	200/300	5	5	8 KB

Finally in case (3), the program will not terminate. Thus

$$Pr[\text{we terminate at } t | t > T, \mathsf{ORAM}(S)_t, T, \text{tamper}] = 0,$$

meaning the entropy is 0. Information can only leak in case (2), and we leak at most 1 bit. Thus, we have shown that a stronger adversary (namely one that knows T a priori) learns at most 1 bit. Therefore, the actual adversary only learns at most 1 bit from our scheme. □

7.6 Generality of Construction

Our integrity verification scheme only requires that the ORAM protocol has a PosMap that randomly remaps each accessed block to a fresh location after each access. Thus, the scheme can also be applied to the works cited in § 5.4. We remark that while Unified ORAM only applies to Recursive position-based ORAM constructions, our integrity scheme can be used regardless of whether Recursion is used.

8 Evaluation

We now describe our hardware prototype of Tiny ORAM on a Virtex-7 VC707 FPGA board and analyze its area and performance characteristics. We show results for our implementation of Unified ORAM (§ 5) and RAW ORAM (§ 6). We leave implementing integrity verification (§ 7) as future work.

8.1 Evaluation Metrics

We evaluate our design in terms of performance and area. Performance is measured as the latency (in cycles or real time) between when a processor requests a block and Tiny ORAM *returns* that block.

Area is calculated in terms of FPGA 'cells' and block RAM and measured post place-and-route (i.e., represent final hardware area numbers). Total cell count is the number of FPGA lookup-tables (i.e., logic gates) plus the number of flip-flops. Block RAM (BRAM for short) are 36 KB SRAM memories on the FPGA.

8.2 Implementation

As described so far, the Frontend (Recursive or Unified ORAM) always communicates to memory through the Backend (Basic Path ORAM or RAW Path ORAM).

Symmetric Encryption. For symmetric encryption, we adopt AES-128 CTR mode as discussed in § 6.4. For actual AES operations, we use a single instance of "tiny aes," a pipelined AES core that is freely downloadable from Open Cores [1]. Tiny aes has a 21 cycle latency and can produce 128 bits of output per cycle. Further, each tiny aes core costs ∼ 6500 FPGA cells and 86 BRAM.[5] To implement the time-multiplexing scheme from § 6.4, we simply add state to track whether tiny aes's output (during each cycle) corresponds to IV_1 or IV_2.

[5] BRAM are used to store the AES SBOX. We have also seen the FPGA tools implement the SBOX in logic, in which case each core costs ∼ 11000 cells and no BRAM.

Parameterization. We study the design variants shown in Table 3. The naming convention is frontend_backend_e{#AES}, where frontend can be a baseline Recursive ORAM or Unified ORAM (§ 5) and backend can be a baseline Path ORAM or RAW ORAM (§ 6). #AES is the number of tiny aes cores used. We use recursive_basic_e3 as a baseline and further split the Unified/RAW ORAM designs into different configurations to show where overheads come from.

All configurations use $B = 512$, $L = 20$ and $H = 3$. We chose $B = 512$ (64 Bytes) to show that Tiny ORAM can run even very small block sizes without imposing hardware performance bottlenecks. We are constrained to set $L = 20$ because this setting fills the VC707's 1 GB DRAM DIMM, but will discuss working set scaling at the end of this Section. For this L, we set $X = 16$ and $H = 3$ as this is sufficient to yield a small on-chip position map ($\sim$ 8 KB) and exploit locality in PosMap blocks. Z is chosen based on what is known to be optimal for both basic Path ORAM and RAW ORAM (§ 6.1).

We did not implement Recursive ORAM without a Unified ORAM frontend (i.e., recursive_basic_e3). We approximate that design's access latency as unified_basic_e3 after disabling the PLB, and subtracting 210 cycles from each access' latency. We subtract cycles because PosMap ORAM lookups in Recursive ORAM will be cheaper than Data ORAM lookups [25].

Clock regions. The DRAM controller on the VC707 board runs at 200 MHz and transfers 512 bits/cycle. To ensure that DRAM is Tiny ORAM's bottleneck, we optimized our design's timing to run at 200 MHz.

DRAM controller. We interface with DDR3 DRAM through a stock Xilinx on-chip DRAM controller. All DDR3 DRAM reads/writes accept 512 bits per cycle. From when a read request is presented to the DRAM controller, it takes $\sim$ 30 FPGA cycles to return data for that read (i.e., without ORAM). The DRAM controller pipelines requests. That is, if two reads are issued in consecutive cycles, two 512 bit responses arrive in cycle 30 and 31.

Practical AES savings. Given this DRAM rate, RAW ORAM requires 2 tiny aes cores, running at 200 MHz, to completely hide mask generation for IV_1 (§ 6.4). To reduce area further, we run the tiny aes core (and a small amount of control logic) at 300 MHz,[6] which reduces the tiny aes core count to 1. Basic Path ORAM requires 3 tiny aes cores clocked at 300 MHz; thus our savings in practice is 3×.

Stash scan penalty. In § 4.2, we introduced the parameter C' as the number of cycles between when an ORAM access starts and when the first data from DRAM arrives at the Stash. Given tiny aes and our FPGA's DRAM read latency, C' is roughly $21 + 30$ cycles (depending on the AES clock frequency, and other small latencies introduced by the implementation). Recall from § 4.2 and § 6.1 that, for $\lambda = 80$, the original Path ORAM backend with $Z = 4$ and RAW Path ORAM backend with $Z = 5$ require $C = 78$ and $C = 64$, respectively. Thus, the Stash scan algorithm from § 4 introduces $78 + 3 - C' = 30$ cycles and $64 + 3 - C' = 16$ cycles per ORAM access, respectively.

Working set scalability. We note that all configurations in Table 3 scale to large working sets with small on-chip storage due to the recursive ORAM construction (§ 3.2). Increasing the working set by a factor of W, while fixing H and X (the levels of PosMap hierarchy and leaves per PosMap block), causes the on-chip PosMap to grow by a factor of W. Alternatively, increasing H by 1 causes the on-chip PosMap to shrink by a factor of X and causes the ORAM access latency to increase by a factor of $(H + 1)/H$ (due to extra PosMap ORAM lookups) in the worst case.

8.3 Access Latency Comparison

Figure 5 shows the average FPGA clock cycle latency for Tiny ORAM to return a data block to a requester, given different program access patterns. All results are collected by feeding traces to a real instance of Tiny ORAM running on hardware.

scan and rand are synthetic patterns that scan memory (i.e., $a, a + 1, a + 2, \dots$) and perform random accesses, respectively. That is, scan has perfect locality whereas rand has the worst locality. We additionally show memory access traces for the SPEC workloads libq, bzip2, gcc and sjeng to show how address locality impacts performance for real programs. All SPEC program traces contain 1 million read operations taken from representative regions of each program's execution.

[6] 300 MHz is close to the FPGA's limit.

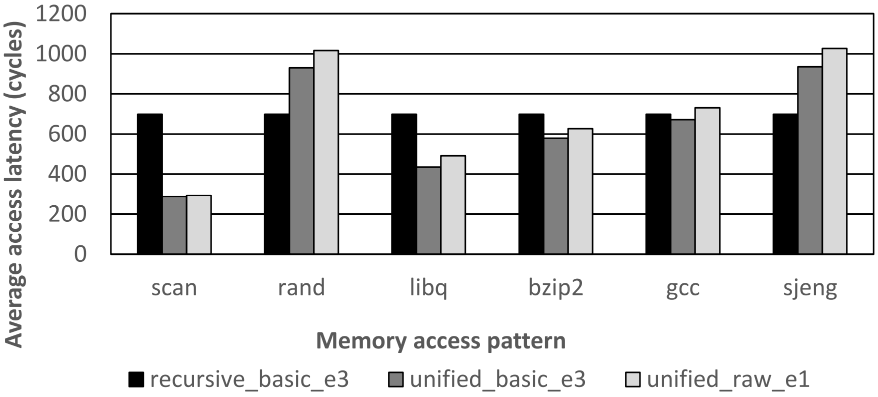

Figure 5: Average number of FPGA clock cycles needed to complete an ORAM access.

Table 4: Design hardware area study. '% FPGA' shows area relative to the VC707 FPGA's capacity.

Configuration	Cell (% FPGA)	BRAM (% FPGA)
unified_basic_e3	39590 (4%)	322 (32%)
unified_raw_e1	31600 (3%)	146 (14%)

We see that program locality is erratic in practice, which shows how different design points with different ORAM block sizes are needed depending on the application. The libq program has good locality and closely approximates scan. Programs with good locality should adopt Unified ORAM (§ 5) and/or larger block sizes (e.g., 4 KB) as used in Phantom [20]. For instance, we see that unified_basic_e3 reduces latency 39% over recursive_basic_e3.

On the other hand, some programs (e.g., sjeng) have bad locality. For these applications, a large block size (e.g., Phantom) hurts performance since most of the data in each block will not be used. Likewise, the PLB will incur a high miss rate, nullifying the benefits from Unified ORAM. The best strategy for these applications is to make the block size as small as possible, and use schemes to minimize the hardware penalty of small blocks (e.g., ideas from § 4 and § 6.4). Specifically, the 64 Byte ($B = 512$) block size that we assume allows Tiny ORAM to return data in $\sim$ 950 cycles (4.75 μs at 200 MHz). We will analytically compare this figure to Phantom by reducing that design's DRAM bandwidth to 512 bits/cycle (to normalize to the VC707 board). In that case, Phantom should be able to fetch a 4 KB block in $27 - 52$ μs (i.e., double their reported access latency), which shows the large speedup potential for small blocks.

8.4 Hardware Area Comparison

We now compare the hardware area for different design variants, shown in Table 4. Our main proposal, that with a Unified frontend and RAW backend, is extremely low area: namely 3%/14% logic/memory. We do not show area for recursive_basic_e3 since we expect it to be very similar to unified_basic_e3 (the 8 KB PLB takes up only 4 BRAM).

The control logic needed to time multiplex tiny aes in RAW ORAM (§ 6.4) is larger than the analogous control logic in basic Path ORAM design and this dampens the AES savings in practice. Despite this, unified_raw_e1 still achieves a 25%/2× reduction in logic/BRAM relative to unified_basic_e3.

We do not compare to Phantom [20] in terms of area because they did not include the cost of encryption in their area.

9 Related Work

Numerous works [11, 12, 13, 15, 17, 32, 14, 29, 9, 30] have significantly improved the theoretical performance of ORAM over the past three decades. Notably among them, Path ORAM [30] is conceptually simple and the most efficient under small client storage. For these reasons, it was embraced by trusted hardware proposals including Tiny ORAM.

Gentry et al. [9] first proposed eviction in reverse lexicographical order and inspired our work. We extended the scheme in several ways. We add a parameter A to reduce evictions, which leads to bandwidth and encryption savings. In parallel to our work, Gentry et al. [10] applied the same idea, still with $A = 1$, to their ORAM construction for use in private database accesses. We also came up with an efficient integrity verification scheme and a simpler Stash analysis for constant bucket size, both of which rely on the deterministic eviction pattern.

Phantom [20] is the first hardware implementation of ORAM, and is most relevant to Tiny ORAM. As mentioned, Phantom implemented basic Path ORAM, and identified several challenges in hardware ORAM design, including efficient stash management, design scalability, and encryption unit area. We address these challenges in this paper.

Ascend [8, 33] is a secure processor proposal that uses ORAM for memory obfuscation and timing protection on top of ORAM [7]. Through simulation, the authors showed that Ascend incurs around $\sim 4\times$ program slowdown and consumes $\sim 6\times$ power compared with an insecure processor.

Ren et al. [25] explored the (recursive) Path ORAM design space through simulation and proposed several optimizations to recursive Path ORAM. We use their optimized recursive Path ORAM as the baseline in our work.

Wang et al. [31] develop mechanisms to reduce the overhead of Recursive ORAM, as does our Unified ORAM scheme. That work only applies in situations when the data stored in ORAM is a part of a common data structure (such as a tree of bounded degree or list). Unified ORAM can be used given any program access pattern. It is worth pointing out, however, that the position map blocks form a tree data structure of bounded degree. Thus, [31] may be able to exploit the same type of locality.

Liu et al. [18] define memory trace obliviousness which can apply to secure processor settings like that proposed in this paper. That work, however, is mostly theoretical at this point and doesn't support modern processors with structures such as caches or branch predictors. Our aim is to support modern processors, and we envision that ORAM should behave and interface to the rest of the processor as an on-chip memory controller. That is, optimizations made to the ORAM controller should not require change to the rest of the chip or to the program running on the chip.

Lorch et al. [19] exploited the parallelism in ORAM operations and used multiple trusted coprocessors to speedup ORAM accesses. Unified RAW Path ORAM also has substantial parallelism, and can adopt their techniques.

10 Conclusion

In this paper we have presented *Tiny ORAM*, a hardware ORAM controller that is both low-latency and low-hardware area, scales to large working sets and supports integrity verification. To achieve these goals, we propose *Unified ORAM* to decrease the overhead of recursive ORAM, *RAW ORAM* to decrease area and simplify theoretical analysis, and a simple yet efficient integrity verification scheme. We demonstrate a working prototype on a Virtex-7 VC707 FPGA board which can return a 64 Byte block in ~ 1.25 μs and requires 3% of the FPGA chip's logic, including the cost of symmetric encryption.

Taken as a whole, our work is an example of how ideas from *circuit design*, *computer architecture* and *cryptographic protocol design*, coupled with *careful engineering*, can lead to significant efficiencies in practice.

Acknowledgments

The authors would like to thank Elaine Shi and Xiao Wang for finding some subtleties in the algorithm and the proof, and the helpful discussions.

References

[1] Open cores. http://opencores.org/.

[2] D. Apon, J. Katz, E. Shi, and A. Thiruvengadam. Verifiable oblivious storage. Cryptology ePrint Archive, Report 2014/153, 2014. http://eprint.iacr.org/.

[3] 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.

[4] 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.

[5] K.-M. Chung, Z. Liu, and R. Pass. Statistically-secure oram with o(log squared n) overhead. *CoRR*, abs/1307.3699, 2013.

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

[7] C. Fletcher, L. Ren, X. Yu, M. Van Dijk, O. Khan, and S. Devadas. Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs. In *Proceedings of the Int'l Symposium On High Performance Computer Architecture*, 2014.

[8] 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.

[9] 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.

[10] C. Gentry, S. Halevi, C. Jutla, and M. Raykova. Private database access with he-over-oram architecture. Cryptology ePrint Archive, Report 2014/345, 2014.

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

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

[13] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, pages 95–100, New York, NY, USA, 2011. ACM.

[14] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Practical oblivious storage. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, CODASPY '12, pages 13–24, New York, NY, USA, 2012. ACM.

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

[16] 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.

[17] 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.

[18] C. Liu, M. Hicks, and E. Shi. Memory trace oblivious program execution. In *Proceedings of the 2013 IEEE 26th Computer Security Foundations Symposium*, CSF '13, 2013.

[19] J. R. Lorch, J. W. Mickens, B. Parno, M. Raykova, and J. Schiffman. Toward practical private access to data centers via parallel oram. *IACR Cryptology ePrint Archive*, 2012:133, 2012. informal publication.

[20] 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.

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

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

[23] 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.

[24] L. Ren, C. Fletcher, X. Yu, M. van Dijk, and S. Devadas. Integrity verification for path oblivious-ram. In *Proceedings of the 17th IEEE High Performance Extreme Computing Conference*, September 2013.

[25] 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.

[26] L. F. G. Sarmenta, M. van Dijk, C. W. O'Donnell, J. Rhodes, and S. Devadas. Virtual Monotonic Counters and Count-Limited Objects using a TPM without a Trusted OS. In *Proceedings of the 1st STC'06*, Nov. 2006.

[27] 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.

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

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

[30] 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.

[31] X. Wang, K. Nayak, C. Liu, E. Shi, E. Stefanov, and Y. Huang. Oblivious data structures. *IACR Cryptology ePrint Archive*, 2014.

[32] 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.

[33] 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.

[34] 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.

A Unified ORAM Pseudo-Code

Algorithm 3 describes the Unified ORAM scheme from § 5. Line 17 uses an easier way to calculate PosMap block address a_i. In § 5.2, we used $a_i = (\Sigma_{j=0}^{i} N/X^j) + a_0/X^i$. It is easy to check the equivalence of the two (assuming X^H divides N):

$$\begin{aligned} a_{i+1} &= (\Sigma_{j=0}^{i+1} N/X^j) + a_0/X^{i+1} \\ &= N + [(\Sigma_{j=0}^{i} N/X^j) + a_0/X^i]/X \\ &= N + a_i/X \end{aligned} \tag{1}$$

Note that all division includes flooring.

The PLB is an ordinary cache that supports three operations, Lookup Remap and Refill. Lookup(a) (Line 13) looks for the content associated with address a, in our case the leaf label of block a. If the leaf exists in the PLB (a 'hit'), it is returned; otherwise (a 'miss') $\perp$ is returned. On a PLB hit, we call Remap(a, l') to change the leaf label of a to l'. Refill (Line 23) happens when there is a PLB miss. This brings the missed content into the PLB, and possibly evicts another block from PLB (Line 24), which needs to be put back into the ORAM tree. In our design, we use an exclusive PLB, meaning that any PosMap block cached in the PLB is not in ORAM. The benefit of an exclusive design was discussed in [25]. To support exclusive PLB, we add another two operations 'read_rmv' and 'append' to the ORAM backend on top of 'read' and 'write'.

A read_rmv operation reads the block and removes it from the ORAM (this is the only difference from a read operation). An append operation simply adds the block to Stash without accessing any path. Whenever UORAMFrontend() requests a PosMap block from the ORAM tree, it uses read_rmv. After the access, the requested PosMap block will be removed from ORAM and added into the PLB, together with the leaf it is mapped to. This usually evicts another PosMap block from from the PLB, which is simply put back into ORAM with its leaf label using append. A read_rmv together with the append that follows has the same effect on stash occupancy as a read operation.

B RAW ORAM pseudo-Code

$RAWCnt$ is the RAW counter, which counts from 0 to $A-1$. Whenever $RAWCnt = 0$, we perform an RW access to the next path in reverse lexicographic order l_g. l_g is simply the lower order L bits of the global RW access counter G, which tracks the total number of RW access we have performed so far. G and $RAWCnt$ persist through all the calls to $\mathsf{RAWORAMBackend}()$.

As mentioned in §6.1, an RW access reads and writes a path, similar to a basic Path ORAM operation except for the difference: it does not return, write or remap any block. Its sole purpose is to evict blocks from Stash to the ORAM tree.

An RO access returns/updates and remaps a data block as requested. It reads the entire path $\mathcal{P}(l)$. For every bucket, it first decrypts its header (Line 23), which contains the addresses of the blocks in this bucket. If a is one of those of addresses (Line 25), it means this bucket contains the requested block. In that case, we decrypt the rest of the bucket and add the requested block to the Stash (Line 26-27). We also need to remove this block from the bucket (Line 28) since its content has changed. Note that this step only affects the header (Line 29). At the end of the access, we re-encrypt and write back the headers for each bucket on path $\mathcal{P}(l)$ (Line 34).

The rest of the steps (append/return/update/remap the block) are unchanged from Path ORAM backend. RAW Path ORAM provides the same interface as Path ORAM to the frontend. Therefore, simply replacing $\mathsf{PORAMBackend}()$ in Algorithm 3 with $\mathsf{RAWORAMBackend}()$ gives the final proposal of this paper, *Unified RAW Path ORAM.*

C RAW ORAM Stash Analysis

In this section we analyze Stash occupancy for a non-recursive RAW Path ORAM. Following the notations in Path ORAM [30], by $\mathsf{ORAM}_L^{Z,A}$ we denote a non-recursive RAW Path ORAM with $L+1$ levels and bucket size Z, and does one RW access per A RO accesses. The root is at level 0 and the leaves are at level L. We define Stash occupancy $\mathsf{st}\left(\mathcal{S}_Z\right)$ to be the number of real blocks in Stash after a sequence of ORAM sequences (this notation will be further explained later). We will prove that $\Pr\left[\mathsf{st}\left(\mathcal{S}_Z\right) > R\right]$ decreases exponentially in R for certain Z and A combinations. As it turns out, the deterministic eviction patter in RAW ORAM simplifies the proof.

Proof Outline. The proof consists of several steps. The first two steps are similar to Path ORAM [30]. We introduce ∞-ORAM, denoted as $\mathsf{ORAM}_L^{\infty,A}$, which has a infinite bucket size and after the post-processing algorithm G has exactly the same distribution of blocks over all buckets and Stash. 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. Then we calculate the average usage of subtrees in ∞-ORAM. Finally, we apply the Chernoff bound on their actual usage to complete the proof.

C.1 ∞-ORAM

The proof for RAW Path ORAM needs Lemma 1 and Lemma 2 for Path ORAM in Stefanov et al. [30], which we restate in this subsection. We notice that the proof for Lemma 2 there is not very detailed, so we make the necessary changes to the proof techniques and give a more rigorous proof here.

We first 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 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 modified greedy post-processing algorithm G, 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[7], G process the blocks in bucket b_i^∞ in the following way:

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

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^∞, 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's 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 $i = 0, 1, \cdots 2^{L+1}$.

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

To prove this lemma, we made a little change to RAW Path ORAM algorithm. In RAW ORAM, an RO access adds the block of interest to 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 to a *stale* block. On a 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. 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 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 leaf l in $\mathsf{ORAM}_L^{Z,A}$ and RW_l^∞ be an RW access to leaf 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

$$\mathcal{S}'_Z = \mathsf{RW}_l^Z(\mathcal{S}_Z) = G_{\mathcal{S}'_Z}\left(\mathsf{RW}_l^\infty(\mathcal{S}_Z)\right) = G_{\mathcal{S}'_Z}\left(\mathsf{RW}_l^\infty\left(G_{\mathcal{S}_Z}(\mathcal{S}_\infty)\right)\right).$$

The last equation is due to the induction hypothesis.

It remains to show that $G_{\mathcal{S}'_Z}\left(\mathsf{RW}_l^\infty\left(G_{\mathcal{S}_Z}(\mathcal{S}_\infty)\right)\right) = G_{\mathcal{S}'_Z}\left(\mathsf{RW}_l^\infty(\mathcal{S}_\infty)\right)$, 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 can also decompose $G_{\mathcal{S}'_Z}$ into $g'_1 g'_2 \cdots g'_{2^{L+1}}$ where each g'_i process bucket $b_i'^\infty$ of $\mathcal{S}'_\infty$ in reference to $b_i'^Z$ of $\mathcal{S}'_Z$. We now show that for any $0 < i < 2^{L+1}$, $G_{\mathcal{S}'_Z}\left(\mathsf{RW}_l^\infty\left(g_1 g_2 \cdots g_i(\mathcal{S}_\infty)\right)\right) = G_{\mathcal{S}'_Z}\left(\mathsf{RW}_l^\infty\left(g_1 g_2 \cdots g_{i-1}(\mathcal{S}_\infty)\right)\right)$. 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 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. Then their order can be swapped, $G_{\mathcal{S}'_Z}\left(\mathsf{RW}_l^\infty\left(g_0 g_1 g_2 \cdots g_i(\mathcal{S}_\infty)\right)\right) = G_{\mathcal{S}'_Z} g_i\left(\mathsf{RW}_l^\infty\left(g_0 g_1 g_2 \cdots g_{i-1}(\mathcal{S}_\infty)\right)\right)$. 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 the RW access to $\mathcal{P}(l)$ since $b_{i/2}$ is the highest intersection (towards leaf) these blocks can go. So g_i can be swapped with RW_l^∞ and can be merged into $G_{\mathcal{S}'_Z}$ similar to 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 that $G_{\mathcal{S}'_Z}\left(\mathsf{RW}_l^\infty\left(G_{\mathcal{S}_Z}(\mathcal{S}_\infty)\right)\right) = G_{\mathcal{S}'_Z}\left(\mathsf{RW}_l^\infty(\mathcal{S}_\infty)\right) = G_{\mathcal{S}'_Z}(\mathcal{S}'_\infty)$ and completes the proof. □

Now we investigate when a ∞-ORAM state $\mathcal{S}_\infty$ will lead to Stash usage of more than R blocks after post-processing. We define rooted subtree, or subtree for short, in the rest of the proof. We say a rooted subtree $T \in \mathsf{ORAM}_L^{\infty,A}$ if T contains the root of $\mathsf{ORAM}_L^{\infty,A}$; in particular, if a node is contained in T, then so are all its ancestors. We define $n(T)$ to be the total number of nodes in T. We define $c(T)$ (the capacity of T) to be maximum number of blocks T can hold; for (RAW) Path ORAM $c(T) = n(T) \cdot Z$. Lastly define the usage $X(T)$ as the actual number of real blocks that are stored in T. The following lemma characterizes Stash usage of ∞-ORAM after post-processing:

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

Proof. If part: Suppose T is a subtree such that $X(T) > c(T) + R$. 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 Stash by G.
Only if part: Suppose that $\mathsf{st}\left(G_{\mathcal{S}_Z}\left(\mathcal{S}_\infty\right)\right) > R$. Define T to be the maximal subtree that contains all buckets with 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 Stash. Hence, all blocks that are in Stash must have originated from T. Therefore, it follows that $X(T) > c(T) + R$. □

By Lemma 1 and Lemma 2, we have

$$\begin{aligned} &\Pr\left[\mathsf{st}\left(\mathcal{S}_Z\right) > R\right] = \Pr\left[\mathsf{st}\left(G_{\mathcal{S}_Z}\left(\mathcal{S}_\infty\right)\right) > R\right] \\ &\le \sum_{T \in \mathsf{ORAM}_L^{\infty,A}} \Pr\left[X(T) > c(T) + R\right] \\ &< \sum_{n \ge 1} 4^n \max_{T:n(T)=n} \Pr\left[X(T) > c(T) + R\right] \qquad (2) \end{aligned}$$

C.2 Average Subtree and Bucket Load

The following lemma will be used in the next subsection:

Lemma 3. *For all 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 average load of T has the following upper bound:*

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

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 $\forall b \in \mathsf{ORAM}_L^{\infty,A}$, $E(Y(b)) \le A/2$. For simplicity, we will write $Y(b)$ as Y.

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), even though 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 $E(Y) \le N \cdot 2^{-L} \le A/2$.

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. With loss of generality, assume $M_1 < M_2$. We then time-stamp the blocks as follows. When a block is accessed and remapped, if m RW accesses have happened, then the block gets time stamp m. Blocks with $m \le M_1$ 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 occurred. Therefore, only blocks with time stamp $M_1 < m \le M_2$ can be in b. There are at most $D = A|M_1 - M_2|$ such blocks [8] 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 RAW ORAM makes it easy to find out that $|M_1 - M_2| = 2^i$. Therefore, $E(Y) \le D \cdot 2^{-(i+1)} = A/2$ for any non-leaf bucket as well. □

[8] 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.

C.3 Chernoff Bound

$X(T) = \sum_i X_i(T)$, where each $X_i(T) \in \{0, 1\}$ and indicates whether the i-th block (can be either real or stale) is in T. $X_i(T)$ is determined by its time stamp i, the leaf label of block i. Thus they are independent from each other, and we can apply Chernoff bound [4].

For simplicity, we write $n = n(T)$ $\hat{c} = c(T) = nZ$, $a = A/2$, $u = E(X(T)) < n \cdot a$ (by Lemma 3) and $\hat{c}/u > Z/a$. Let $\delta = (\hat{c} + R - u)/u$. By Chernoff bound,

$$\begin{aligned} \Pr\left[X(T) > \hat{c} + R\right] &= \Pr\left[X(T) > (1+\delta)u\right] \\ &\le e^{[\delta - (1+\delta)\ln(1+\delta)]u} \\ &\le e^{\hat{c}+R-u-(\hat{c}+R)\ln\left(\frac{\hat{c}+R}{u}\right)} \\ &< e^{R[1-\ln(\hat{c}/u)]} \cdot e^{-\hat{c}[\ln(\hat{c}/u)-1]-u} \\ &< (ea/Z)^R \cdot e^{-n[Z\ln(Z/ae)+a]} \end{aligned} \tag{3}$$

In the last step, note that $\hat{c}[\ln(\hat{c}/u) - 1] + u$ decreases with u when $u < \hat{c} = nZ$ and that $u < na$. So it holds for $Z \geq a = A/2$.

Now we will choose Z and A such that $ea/Z < 1$ and $Z\ln(Z/ae) + a > \ln 4$. If these two conditions hold, from (2) we have,

$$\Pr\left[\mathsf{st}\left(\mathcal{S}_Z\right) > R\right] = \sum_{n \geq 1} \alpha^R \cdot \beta^n < \frac{\alpha^R}{1-\beta}$$

for some $0 < \alpha, \beta < 1$.

Trying out different Z and A, we get some working configurations, among which Z3A1, Z5A3, Z6A4, Z7A5 are several competitive ones.

C.4 Extension: Dummy Accesses

From experiments, we find that the most competitive configurations are Z2A1, Z3A2, Z4A3, Z5A5 §6.1). Unfortunately, the current analysis is not very tight and does not cover the above settings, which will be actually in use. In this section, we introduce the technique of adding dummy accesses to get very close to the above competitive parameters.

The idea is to interpret the RW frequency A in a probabilistic sense such that we do not have to restrict A to be an integer. For example, if for every access, we make it an RO access with 80% probability and an RW access with 20%. This would be equivalent to $A = 4$ in the proof of Lemma 3 and the Chernoff bound. Therefore, by adjusting the probability of RO accesses vs. RW accesses, we can have make A to be any rational number. Plugging non-integer A into the Chernoff bound, we see Z2A0.92, Z3A1.9, Z4A2.9 have negligible Stash overflow probability. These three are very close to some of the experimentally competitive parameters mentioned above.

Algorithm 3 Unified Path ORAM frontend.

```
Inputs: Address $a$, Operation $op$, Write Data $D'$
function AccessORAM($a, op, D'$)
    $l, l' \leftarrow$ UORAMFrontend($a$)
    return PORAMBackend($a, l, l', op, D'$)
function UORAMFrontend($a$)
    $a_0 \leftarrow a$
    for $h \leftarrow 0$ to $H - 1$ do
        $l'_h \leftarrow$ PRNG() $\mod 2^L$
        if $h \overset{?}{=} H - 1$ then
            $l_h \leftarrow$ PosMap[$a_h$]                          ▷ will always hit
            PosMap[$a_h$] $\leftarrow l'_h$
        else
            $l_h \leftarrow$ PLB.Lookup($a_h$)                      ▷ may miss
            if $l_h \overset{?}{\sim=} \perp$ then                  ▷ hit, start access
                PLB.Remap($a_h, l'_h$)
                break
        $a_{i+1} \leftarrow a_i + N/X$                              ▷ Equation 1
    for $i \leftarrow h$ to 1 do                                    ▷ PosMap block accesses
        $D_i \leftarrow$ PORAMBackend($a_i, l_i, l'_i$, read_rmv, $\perp$)
        $j \leftarrow a_{i-1} \mod X$                               ▷ $l_{i-1}$ is $j$-th leaf in $D_i$
        $l_{i-1} \leftarrow D_i[j]$
        $D_i[j] \leftarrow l'_{i-1}$
        $(a_e, l_e, D_e) \leftarrow$ PLB.Refill($a_i, l_i, D_i$)
        if $a_e \overset{?}{\sim=} \perp$ then                      ▷ PLB eviction
            PORAMBackend($a_e, \perp, l_e$, append, $D_e$)
    return $l_0, l'_0$
function PORAMBackend($a, l, l', op, D'$)
    if $op \overset{?}{=}$ append then
        InsertBlocks(Stash, $(a, l', D')$)                          ▷ append and return
        return
    ReadPath($l$)                                                   ▷ same as in Algorithm 1
    $r \leftarrow$ FindBlock(Stash, $a$)
    $(a, l, D) \leftarrow$ Stash[$r$]
    if $op \overset{?}{=}$ write then
        Stash[$r$] $\leftarrow (a, l', D')$
    else if $op \overset{?}{=}$ read then
        Stash[$r$] $\leftarrow (a, l', D)$
    else if $op \overset{?}{=}$ read_rmv then                       ▷ remove block
        Stash[$r$] $\leftarrow \perp$
    $\mathcal{S} \leftarrow$ PushToLeaf(Stash, $l$)
    WritePath($l, \mathcal{S}$)                                     ▷ same as in Algorithm 1
    return $D$
```

Algorithm 4 RAW Path ORAM backend.

```
Initial: $RAWCnt = 0$, $G = 0$
function RAWORAMBackend($a, l, l', op, D'$)
    $RAWCnt \leftarrow RAWCnt + 1 \mod A$                          ▷ RAW counter
    if $RAWCnt \overset{?}{=} 0$ then
        $l_g \leftarrow G \mod 2^L$
        RWAccess($l_g$)
        $G \leftarrow G + 1$
    return ROAccess($a, l, l', op, D'$)
function RWAccess($l$)
    for $i \leftarrow 0$ to $L$ do                                  ▷ read path
        bucket $\leftarrow$ Decrypt$_K$($\mathcal{P}(l)[i]$)
        InsertBlocks(Stash, bucket)
    $\mathcal{S} \leftarrow$ PushToLeaf(Stash, $l$)
    for $i \leftarrow 0$ to $L$ do                                  ▷ write path back
        bucket $\leftarrow \mathcal{S}[i * L, \dots, i * L + Z - 1]$
        RemoveBlocks(Stash, bucket)
        $\mathcal{P}(l)[i] \leftarrow$ Encrypt$_K$(bucket)
function ROAccess($a, l, l', op, D'$)
    if $op \overset{?}{=}$ append then
        InsertBlocks(Stash, $(a, l', D')$)                          ▷ append and return
        return
    for $i \leftarrow 0$ to $L$ do                                  ▷ read path
        header[$i$] $\leftarrow$ Decrypt$_K$($\mathcal{P}(l)[i]$.header)   ▷ decrypt header
        for $j \leftarrow 0$ to $Z - 1$ do
            if header[$i$].addr[$j$] $\overset{?}{=} a$ then           ▷ found block $a$
                bucket $\leftarrow$ Decrypt$_K$($\mathcal{P}(l)[i]$)
                InsertBlocks(Stash, bucket[$j$])
                bucket[$j$] $\leftarrow \perp$
                header[$i$] $\leftarrow$ bucket.header

    Access the block ... Same as line 32-39 in Algorithm 3.

    for $i \leftarrow 0$ to $L$ do                                  ▷ re-encrypt and writeback header
        $\mathcal{P}(l)[i]$.header $\leftarrow$ Encrypt$_K$(header[$i$])
    return $D$
```