RAW Path 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

June 4, 2014

Abstract

We propose *RAW Path ORAM*, an ORAM construction that improves the state of the art Path ORAM in several ways. First, RAW Path ORAM reduces the amount of encryption operations by 4× compared with Path ORAM. Second, RAW Path ORAM enables a much more efficient and simpler integrity verification scheme. Third, RAW Path ORAM dramatically simplifies the theoretical analysis on client storage requirement (stash size).

We build RAW Path ORAM in hardware and name it *Tiny ORAM*. Tiny ORAM is the first hardware ORAM design that efficiently supports small client storage, arbitrary block sizes (e.g., 64 Bytes to 4096 Bytes) and integrity verification. Block size flexibility allows Tiny ORAM to greatly reduce the worst-case access latency for ORAM running programs with erratic data locality. To reduce the performance overhead that comes with small client storage, we add *Unified ORAM* scheme that further decreases ORAM access latency by up to 39% on real workloads.

We demonstrate a complete working prototype on a stock FPGA board. Tiny ORAM requires 5%/15% of the FPGA logic/memory (including encryption and integrity verification) and can complete an ORAM access for a 64 Byte block in $1.25 - 4.75\mu s$.

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 [32] 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, 18, 30, 26, 29].

An important use case for ORAM is in *trusted hardware* [20, 27, 29, 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, makes asymptotic improvements over prior work [20]. Our scheme can support any reasonable block size (e.g., from 64-4096 Bytes) without introducing performance bottlenecks.

***Unified ORAM* to efficiently provide working set scalability (§ 5).** For the first time, we build the recursive ORAM construction of Shi et al. [26] into hardware. We then propose a new *Unified ORAM* scheme that reduces ORAM access latency by up to 39% on real workloads, when compared to a baseline Path ORAM [29] design that uses recursion.

***RAW ORAM* to reduce design area footprint (§ 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.

Integrity verification. PWP also enables a novel and efficient integrity verification scheme (§ 6.3). To our knowledge, we are the first to build any integrity scheme for ORAM in hardware. We further propose algorithmic optimizations to reduce the number of required hash units by $4\times$.

Novel and simple proof technique. We prove that RAW Path ORAM has negligible stash overflow probability under certain configurations. Our proof is much simpler than that of Path ORAM.

Real implementation and evaluation (§ 8). We implement *all* aspects of the above ideas 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 [29], 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. We will release the source code to the community if the paper is accepted.

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 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 $\overleftarrow{a}$, let $\mathsf{ORAM}(\overleftarrow{a})$ be the resulting randomized data request sequence of an ORAM algorithm. Each element in a data request sequences follows the standard RAM interface, i.e., is a (address, op, write data) tuple. ORAM privacy requires that for any $\overleftarrow{a}$ and $\overleftarrow{a'}$, $\mathsf{ORAM}(\overleftarrow{a})$ and $\mathsf{ORAM}(\overleftarrow{a'})$ are computationally indistinguishable if $|\mathsf{ORAM}(\overleftarrow{a})| = |\mathsf{ORAM}(\overleftarrow{a'})|$.

The standard definition (mostly used in theoretical works) is that if $|\overleftarrow{a}| = |\overleftarrow{a'}|$, the resulting ORAM sequences should be indistinguishable. This usually means that $|\mathsf{ORAM}(\overleftarrow{a})|$ is completely determined (and thus revealed) by $\overleftarrow{a}$. 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 $\overleftarrow{a}$ 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, $|\mathsf{ORAM}(\overleftarrow{a})|$ is now determined by, and thus reveals, the number of Last-Level-Cache (LLC) misses in $\overleftarrow{a}$, but not $|\overleftarrow{a}|$. If an adversary knows $|\overleftarrow{a}|$, 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. Following Phantom, 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

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)
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

occur at data-independent times. Also following Phantom, we do not obfuscate *when* an ORAM access is made or the time it takes the program to terminate. These two timing channels have been addressed for Path ORAM by Fletcher et al. [7].

3 Background

As did Phantom, Tiny ORAM originates from Path ORAM [29]; 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 (denoted C, typically 100 to 200). 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 PORAMFrontend() and 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 (decrypt() and encrypt()) use a session key K controlled by the trusted processor. The 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 bucket additionally stores some *header* information (referred to henceforth as the *bucket header*). This state includes each block's current leaf (L bits) and program address (U bits[1]), as well as an initialization vector for symmetric encryption (whose width is a security parameter denoted λ_t). We denote dummy blocks as $\perp$ and also refer to their program address as $\perp$.

Algorithm 1 Basic Path ORAM access.

```
1:  Inputs: Address a, Operation op, Write Data D'
2:  function AccessORAM(a, op, D')
3:      l, l' ← PORAMFrontend(a)
4:      return PORAMBackend(a, l, l', op, D')
5:  function PORAMFrontend(a)
6:      l' ← PRNG_K() mod 2^L
7:      l ← PosMap[a]
8:      PosMap[a] ← l'                                   ▷ remap block
9:      return l, l'
10: function PORAMBackend(a, l, l', op, D')
11:     ReadPath(l)
12:     r ← FindBlock(Stash, a)                          ▷ r points to block a
13:     (a, l, D) ← Stash[r]
14:     if op ?= write then
15:         Stash[r] ← (a, l', D')
16:     else if op ?= read then
17:         Stash[r] ← (a, l', D)
18:     S ← PushToLeaf(Stash, l)                         ▷ see § 4
19:     WritePath(l, S)
20:     return D
21: function ReadPath(l)
22:     for i ← 0 to L do                                ▷ read path
23:         bucket ← Decrypt_K(P(l)[i])
24:         InsertBlocks(Stash, bucket)
25: function WritePath(l, S)
26:     for i ← 0 to L do                                ▷ write path back
27:         bucket ← S[i * L, ..., i * L + Z − 1]
28:         RemoveBlocks(Stash, bucket)
29:         P(l)[i] ← Encrypt_K(bucket)
```

PushToLeaf(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, PushToLeaf(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.

[1]We approximate $U = L$ for the rest of the paper for simplicity. In practice, U may be several bits larger than L.

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. [29] proves that Stash capacity can be bounded to be small if $Z \geq 5$.

There are two security parameters, namely $\lambda = |K|$ where K is the session key and λ_t (described below). We assume $\lambda = 128$ for the rest of the paper, and claim resistance against attacks of computational complexity up to 2^{λ}. We set $\lambda_t = 64$. λ_t is the amount of entropy in initialization vectors used for symmetric encryption. For example, each time a given chunk of data in ORAM is re-encrypted, the initialization vector used for that operation will be incremented by 1 (see § 7.1). Thus, it is only important for the number of ORAM accesses to not exceed 2^{λ_t} and $\lambda_t = 64$ is sufficient for this purpose.

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. [26] 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), 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. We note that this, and later, division operations are rounded down (floored) to the nearest integer. 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.

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, $\mathsf{PushToLeaf}()$ is a challenge for efficient Path ORAM hardware designs. Conceptually, $\mathsf{PushToLeaf}(\mathsf{Stash}, l)$ needs to push every block in 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 Stash to grow uncontrollably.

On the other hand, functionally correct implementations of $\mathsf{PushToLeaf}()$ can cause serious hardware performance bottlenecks due to their computational complexity. For example, a naïve implementation is to scan Stash for each location on path $\mathcal{P}(l)$, which takes $O(C)$ cycles per block. Gentry et al. [9] suggest an $O(L)$ method. Phantom proposed a *heap sort*-based Stash management, reducing this to $O(\log C)$ 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 heap sort design takes 11 cycles to evict a block [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.

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:      S ← {⊥ for i = 0, ..., (L+1) * Z − 1}
4:      Occupied ← {0 for i = 0, ..., L}
5:      for i ← 0 to C − 1 do
6:          (a, l_i, D) ← Stash[i]                          ▷ Leaf assigned to i-th block
7:          level ← PushBack(l, l_i)
8:          if level > −1 then
9:              offset ← level * Z + Occupied[level]
10:             S[offset] ← (a, l_i, D)
11:             Occupied[level] ← Occupied[level] + 1
12:     return S
13: function PushBack(l, l')
14:     t_1 ← (l ⊕ l')||0                                   ▷ Bitwise XOR
15:     t_2 ← t_1 & −t_1                                    ▷ Bitwise AND, 2C negation
16:     t_3 ← t_2 − 1                                       ▷ 2C subtraction
17:     full ← {(Occupied[i] ≟ Z) for i = 0 to L}
18:     t_4 ← t_3 & ∼full                                   ▷ Bitwise AND/negation
19:     t_5 ← reverse(t_4)                                  ▷ Bitwise reverse
20:     t_6 ← t_5 & −t_5
21:     t_7 ← reverse(t_6)
22:     if t_7 ≟ 0 then
23:         return −1                                       ▷ Block is stuck in Stash
24:     return log_2(t_7)                                   ▷ Note: t_7 must be one-hot
```

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 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 $000\cdots111$, 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.

Hardware details. reverse() costs no additional logic and the other bit operations (including $\log_2(x)$ when x is one-hot) are simple combinational logic. The most expensive operations latency-wise are two's complement arithmetic of $(L+1)$-bit words. To meet our FPGA's clock frequency, we had to add 2 pipeline stages after Lines 15 and 16. Crucially, however, a new call to PushBack() can be issued *every* cycle, making amortized throughput 1 cycle/block.

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.

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

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

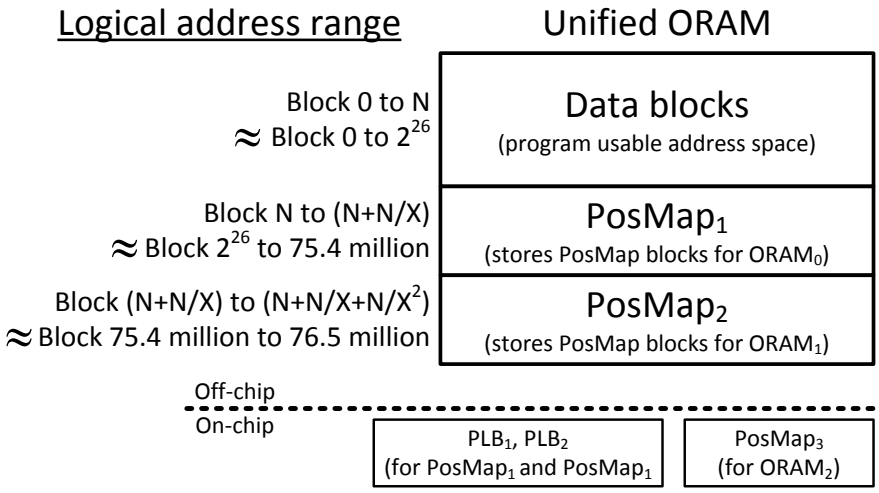

Figure 1: Unified ORAM address space assuming there are $N = 2^{26}$ data blocks and $X = 8$.

$\{\mathsf{PosMap}_1, \mathsf{PosMap}_2, \cdots, \mathsf{PosMap}_{\mathsf{H}-1}\}$, 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 remark that unified ORAM's security only holds under the security definition in § 2. Similar to on-chip cache hit and miss rates, PLB hit and miss rates also affect the ORAM sequence length $|\mathsf{ORAM}(\overleftarrow{a})|$. Now $|\mathsf{ORAM}(\overleftarrow{a})|$ is determined by, and thus reveals, the sum of LLC misses and PLB misses. Every LLC miss turns into a data block access and every PLB miss turns into a PosMap block access.

Under the security definition of this paper, the security of unified ORAM reduces to the security of the backend ORAM. The basic operation of the unified ORAM is to send requests to backend (in the case of Path ORAM backend, each operation reads and writes a random path). Which block is accessed or whether it is a data block or a PosMap block is protected by backend.

Unified ORAM impacts PORAMFrontend() from Algorithm 1 only. Besides Path ORAM, it works with some other ORAM constructions that provide the same interface as PORAMBackend(), including [26, 28, 9], and also *RAW Path ORAM*, which we present in the next section. These constructions all have large client-side storage, and all need recursion to be used in trusted hardware.

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 enables our novel integrity verification scheme. The RAW algorithm impacts 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 is given in Appendix B.

6.1 Overview

Parameter A. We introduce a new parameter A, set at system boot time. RAW Path ORAM splits 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.

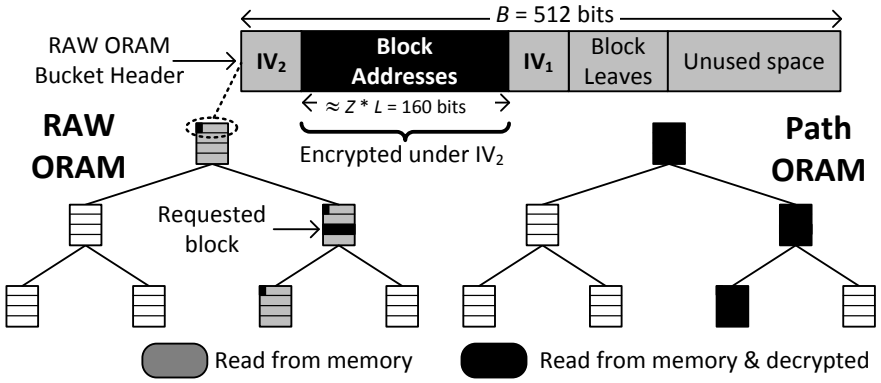

Figure 2: Data read vs. data decrypted on a RAW ORAM RO read (left) and Path ORAM access (right). IV_1 and IV_2 are initialization vectors used for encryption.

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 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 block address to $\perp$ or clearing a block valid bit), 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 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 § 7.1.

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, 29].

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. Stash size does not include the transient storage for the incoming path on an RW access.

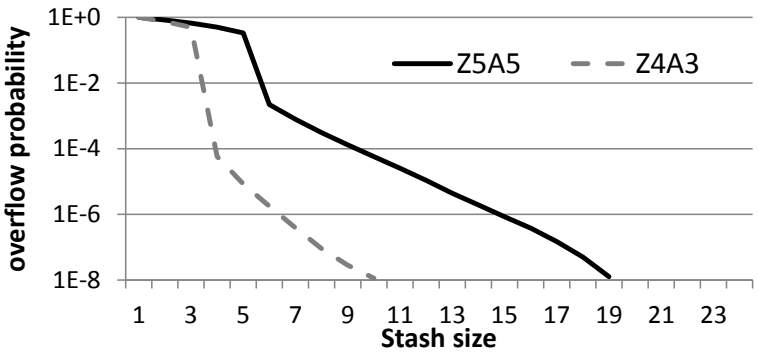

Figure 3: For RAW ORAM with $A=5, Z=5$ and $A=4, Z=3$, experiments show that Stash overflow probability drops exponentially with Stash size. We experiment with $L=25$, simulate for over 1 billion RO accesses, and sample the stash occupancy *before* each RW access.

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. 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.

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 $G \mod 2^L$. Intuitively, reverse lexicographic order improves eviction quality by load-balancing between paths.

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 Integrity Verification

With the write count of every bucket known at all times, we no longer need a Merkle tree for integrity verification as in [24]. Instead, we can simply use a Message Authentication Code (MAC) for each bucket alongside its bucket index and write count.

For a bucket bkt, let $d(\mathsf{bkt})$ be the plaintext data of the bucket, including valid bits, addresses, leaf labels and payloads for all the Z blocks. Let $id(\mathsf{bkt})$ be the bucket's unique index (e.g., physical address in DRAM), and $v(\mathsf{bkt})$ be the write count of the bucket. We compute the following hash and store it in the bucket header:

$$h = \mathsf{MAC}_K(\ d(\mathsf{bkt}) \ ||\ id(\mathsf{bkt}) \ ||\ v(\mathsf{bkt})\ ),$$

where MAC_K can be any Message Authentication Code scheme under secret key K.

On an RW access, we check whether the above MAC matches for each bucket read from the ORAM tree, and also generate the MAC for each bucket to be written out to the ORAM tree. On an RO access, we only check and recompute the MAC for the bucket of interest (it needs to be recomputed because a certain block's address in the bucket will be set to $\perp$). All MACs are re-encrypted to hide which one, if any, is recomputed.

Saving compared with Merkle tree. To integrity-verify Path ORAM using a Merkle tree [24], each access must check and rehash all $L + 1$ buckets on the path. With RAW ORAM, for every A frontend accesses, we need to check and rehash $A + L + 1$ buckets. Assuming $A = 5$ and $L = 20$, this is at least a $4\times$ reduction in the amount of bits hashed. Furthermore, the hash chaining in the Merkle tree solution is inherently serialized when generating the hash. With a small block size and high memory bandwidth, writing a bucket to DRAM only takes $Z + 1$ cycles (§ 8.2). Hash latency will introduce considerable latency to the critical path of each ORAM access no matter how many hash engines we put on-chip. Our integrity verification for RAW ORAM does not have this serialization overhead.

6.4 RAW ORAM Security

Privacy. 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. Interestingly, the deterministic write pattern (PWP) also enables a much simpler proof. We give the proof in Appendix C.

Integrity. Breaking our integrity verification scheme for RAW ORAM is as hard as breaking the underlying MAC. Note that only the $d(\mathsf{bkt})$ come from the ORAM tree; $id(\mathsf{bkt})$ and $v(\mathsf{bkt})$ are computed inside

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

the ORAM controller on each access. Thus an adversary cannot tamper with $id(\mathsf{bkt})$ or $v(\mathsf{bkt})$. To come up with a forgery, the adversary has to find a pair (h', d') such that $h' = \mathsf{MAC}_K(\ d' \ ||\ id(\mathsf{bkt}) \ ||\ v(\mathsf{bkt})\ )$. If the adversary succeeds in doing so, it has broken the underlying MAC scheme.

7 Area Savings In Practice

Despite RAW ORAM's theoretic area savings for encryption and hash units, careful engineering is needed to prevent that savings from turning into performance loss. The basic problem is shown in Figure 4 using encryption as an example. The same issue is present for the hash units in our integrity verification scheme.

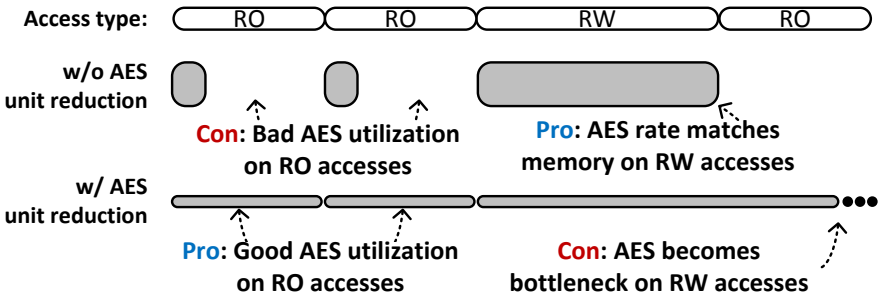

Figure 4: Potential RAW ORAM performance bottleneck. AES refers to symmetric encryption.

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.

This section describes how to get the best of both worlds: maximum utilization (and therefore reduction) of hardware units on RO accesses and no performance bottleneck on RW accesses.

7.1 Symmetric Encryption

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 bucket bkt, as done on an RW ORAM path read, we XOR it with the following mask:

$$\mathsf{AES}_K(v(\mathsf{bkt}) \ ||\ id(\mathsf{bkt}) \ ||\ i)$$

where $v(\mathsf{bkt})$ and $id(\mathsf{bkt})$ are the bucket write count and index as in § 6.3. 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, $v(\mathsf{bkt})$ takes the place of IV_1 and since $v(\mathsf{bkt})$ 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 $\mathsf{AES}_K(\mathrm{IV}_2 \ ||\ id(\mathsf{bkt}) \ ||\ i)$, where IV_2 is stored externally as part of each bucket's header.

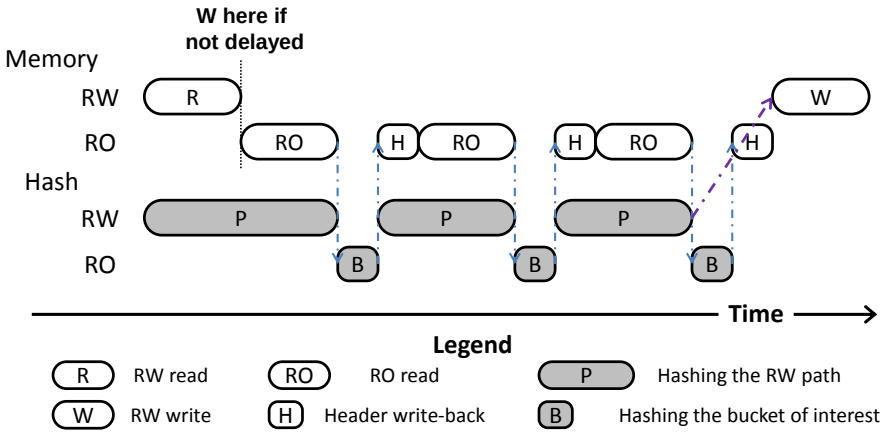

Figure 5: Delayed RW path write-back to make more time for hashing. This figure assumes $A = 3$. Arrows indicate dependencies. Hashing can only start after the path is read. Write-back (path or header) can only happen after hash computation finishes.

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.

7.2 Cryptographic Hashing

Integrity verification has the same performance problem as AES: RW accesses require much higher bandwidth from hash engines than RO accesses. On an RW access, we need to integrity check each bucket on the incoming path and generate hashes for the output path. On an RO access, we only need to check and update the hash for the bucket of interest. Unfortunately, unlike encryption, there is no way to predict the next input to the HMAC. To achieve area savings for HMAC hash units, we propose a *delayed write-back* procedure for RW access which effectively performs RW hash operations in the background.

Figure 5 captures the idea and gives a timeline. After an RW path read, we do not immediately perform the path write-back. Instead, we go on to perform A RO accesses and their header write-backs. Finally, we perform the RW path write-back right before the next RW path read. In this way, we overlap the hashing of the RW path with the A RO accesses, and get more time to hash the RW path. During the process, the hash engines prioritize buckets of interest coming in on RO accesses. Upon completion of hashing buckets of interest, they resume the work of hashing the RW path.

Readers may notice that with delayed RW write-back, the contents in the external ORAM tree are not always fresh. If an RO access reads a path that intersects with the previous RW path—in fact, any two paths intersect at the root—the data we get back from the ORAM tree would be stale. To address this problem, we add a *Coherence Controller* (CC) between the encryption units and Stash.

At a high level, the CC buffers incoming and outgoing RW paths for the integrity verifier. CC handles the RW write-back but ensures the rest of the system always sees the up-to-date data, as Figure 6 shows. Suppose the last RW access is to path $\mathcal{P}(l_g)$. On every RO access to path $\mathcal{P}(l)$, CC determines the intersection of $\mathcal{P}(l_g)$ and $\mathcal{P}(l)$. For buckets that are on both path $\mathcal{P}(l)$ *and* $\mathcal{P}(l_g)$, the fresh copy is in the CC buffer; for buckets that are not on path $\mathcal{P}(l_g)$, their most recent version is in the ORAM tree. CC stitches the two parts together and passes the resulting fresh path to Stash. CC also passes the bucket of interest, if there is any, to integrity verifier. At the end of each RAW round, CC writes the delayed RW path with each bucket's hash back to the ORAM tree.

8 Evaluation

We now describe our complete hardware prototype of Tiny ORAM on a Virtex-7 VC707 FPGA board and analyze its area and performance characteristics.

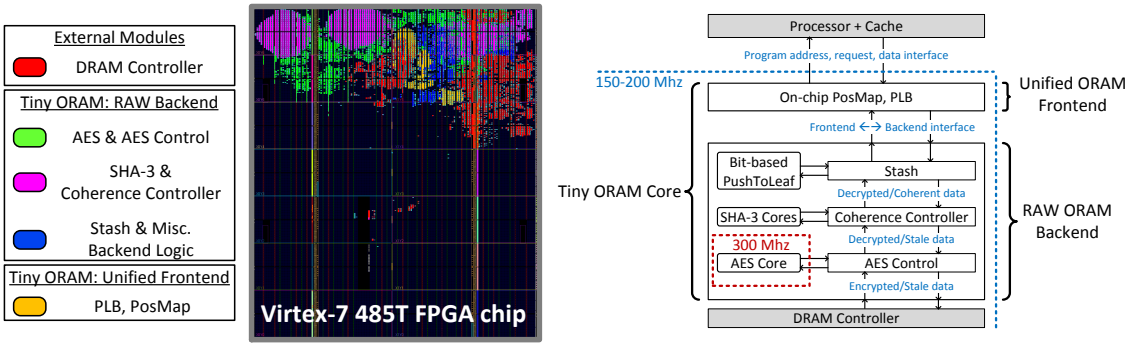

Figure 6: FPGA implementation results and block-level design for unified_raw_e1_h2.

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

Our design at the block and post place-and-route level is shown in Figure 6. 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 § 7.1. 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 $\sim$ 6500 FPGA cells and 86 BRAM.[4] To implement the time-multiplexing scheme from § 7.1, we simply add state to track whether tiny aes's output (during each cycle) corresponds to IV_1 or IV_2.

Cryptographic Hashing. We use a SHA3-512 core, also from Open Cores [1], which we configure as an HMAC by prepending the λ-bit session key to each hash input [2]. The core itself has a throughput of 576-bits in 11 cycles, and a latency of 12 cycles after receiving the entire input. Each core's area is 8805 cells and no BRAM.

We truncate each SHA3-512 digest to 128-bits so that it may fit in each bucket header's unused space (§ 6.1), and note that a 128-bit hash matches the other security parameters in the system. We note that SHA3-512 provides more security than needed; SHA3-224 or SHA3-256 should be acceptable as well. We use SHA3-512 because it was available in Verilog-2001. If one were to obtain and port the SHA3-256 from [17], the expected area savings is 2×, normalized to bandwidth.

Parameterization. We study the design variants shown in Table 2. The naming convention is frontend_backend_e{#AES}_h{#SHA3}, where frontend can be a baseline Recursive ORAM or Unified ORAM

[4]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 $\sim$ 11000 cells and no BRAM.

Table 2: 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
unified_raw_e1_h2	150/300	5	5	8 KB

(§ 5) and backend can be a baseline Path ORAM or RAW ORAM (§ 6). #AES/#SHA3 is the number of tiny aes/SHA3-512 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. unified_raw_e3_h2 is the only configuration with integrity verification.

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. When we add integrity verification, we could not meet the 200 MHz constraint due to the increase in FPGA area and thus run that configuration at 150 MHz.

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 (§ 7.1). To reduce area further, we run the tiny aes core (and a small amount of control logic) at 300 MHz,[5] 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×.

Practical SHA3 savings. The SHA3-512 hash engine can hash a bucket in 70 cycles, and it takes only $\sim$ 6 cycles ($Z+1$) to read one bucket from DRAM. If we do not use delayed path write-back (§ 7.2), we need 12 such hash engines to match DRAM bandwidth. (A design based on a Merkle tree can do no better than this.) With the delayed write-back design, we analytically determined that we only need 3 hash engines, achieving the 4× theoretical saving. Due to additional cycle delays in our real design, however, each round of RO/RW accesses takes longer than expected. Thus, we only need 2 hash engines to hide the RW hash latency and will use this setting for measurement purposes.

Working set scalability. We note that all configurations in Table 2 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 7 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. We note that *absolute time per access* for unified_raw_e1_h2 will increase by

[5] 300 MHz is close to the FPGA's limit.

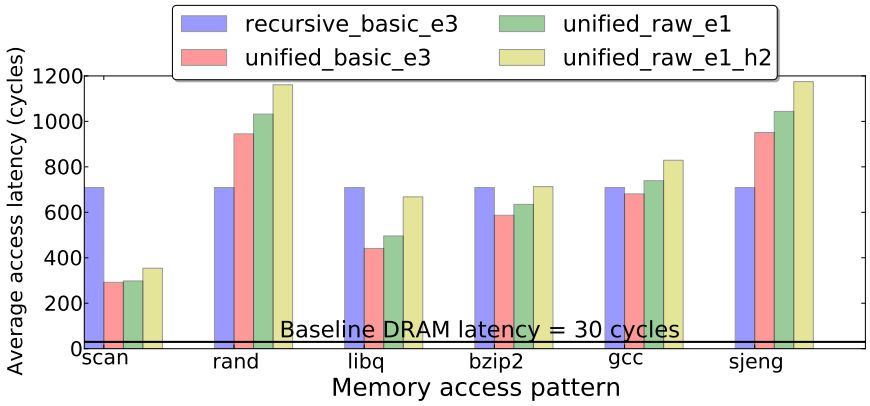

Figure 7: Average number of FPGA clock cycles needed to complete an ORAM access.

33% because we run that configuration at 150 MHz (§ 8.2). For comparison purposes, it takes ~ 30 FPGA cycles to read a single 512-bit burst from DRAM (i.e., without ORAM); after that, 512 bits arrive each cycle.

scan and rand are synthetic patterns that scan memory (i.e., $a, a+1, a+2, \ldots$) 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.

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 § 7). Specifically, the 64 Byte ($B = 512$) block size that we assume allows Tiny ORAM to return data in ~ 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 3. Our main proposals, those with a Unified frontend and RAW backend, are all extremely low area: namely 3%/14% logic/memory without integrity checking and 5%/15% with integrity checking. 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 (§ 7.1) 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.

Table 3: 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%)
unified_raw_e1_h2	52750 (5%)	150 (15%)

9 Related Work

Numerous works [11, 12, 13, 15, 18, 30, 14, 28, 9, 29] have significantly improved the theoretical performance of ORAM over the past three decades. Notably among them, Path ORAM [29] 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, 31] 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.

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 enable integrity checking, and various other optimizations to make our theoretical improvements translate to practice. We demonstrate a complete working prototype on a Virtex-7 VC707 FPGA board; our design can return a 64 Byte block in ~ 1.25 μs and requires 5% of the FPGA chip's logic, including the cost of symmetric encryption and integrity verification.

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] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Keccak sponge function family main document. *Submission to NIST (Round 2)*, 3, 2009.

[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] M. Knezevic, P. Schaumont, A. Satoh, K. Sakiyama, and K. Ota. How can we conduct fair and consistent hardware evaluation for sha-3 candidate?, the second sha-3 candidate conference by nist, 2010.

[18] 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.

[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] 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.

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

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

[29] 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.

[30] 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.

[31] 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.

[32] 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 RAWORAMBackend().

As mentioned in §6.1, an RW access reads and writes a path, similar to a basic Path ORAM operation except for two differences. First, it does not return, write or remap any block, since its sole purpose is to evict blocks from Stash to the ORAM tree. Second, an RW access in RAW ORAM also performs integrity checking for every bucket it reads in (Line 12), and computes the HMAC for every bucket it writes out (Line 18). The HMAC is part of the bucket header, and is encrypted before writing to the ORAM tree. Details of our integrity verification scheme are in § 6.3.

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 25), which contains the addresses of the blocks in this bucket. If a is one of those of addresses (Line 28), it means this bucket contains the requested block. In that case, we decrypt the rest of the bucket, verify its integrity, and add the requested block to Stash (Line 29-31). We also need to remove this block from the bucket (Line 32), and re-MAC the bucket since its content has changed (Line 33). Note that these two steps only affect the header (Line 34): removing the block only involves removing address a from the bucket header and the HMAC is part of the bucket header as well. At the end of the access, we re-encrypt and write back the headers for each bucket on path $\mathcal{P}(l)$ (Line 39).

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 PORAMBackend() in Algorithm 3 with RAWORAMBackend() gives the final proposal of this paper, *unified RAW Path ORAM*.

C RAW ORAM Stash Analysis

In this section we will analyze the stash occupancy for a non-recursive RAW Path ORAM. Following the notations in Path ORAM [29], by $\mathsf{ORAM}_L^{Z,A}$ we denote a non-recursive RAW Path ORAM with $L+1$ levels with 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. $\mathsf{st}\left(\mathsf{ORAM}_L^{Z,A}\right)$ is defined to be the number of blocks in Stash after a sequence of load/store operations, assuming an infinite Stash. We prove that $\Pr\left[\mathsf{st}\left(\mathsf{ORAM}_L^{Z,A}\right) > R\right]$ decreases exponentially in R for certain Z and A combinations.

Proof Outline. The proof consists of several steps. The first two steps are similar to Path ORAM [29]. We introduce ∞-ORAM, denoted as $\mathsf{ORAM}_L^{\infty,A}$, which has a infinite bucket size and after the post-processing algorithm G_Z has exactly the same distribution of blocks over all buckets and Stash. The Stash usage of ∞-ORAM after post-processing is greater than R if and only if there exists a subtree T in ∞-ORAM whose "usage" is more than its "capacity". Then, we calculate the average usage of subtrees in ∞-ORAM and apply the Chernoff bound on their actual usage to complete the proof.

C.1 ∞-ORAM

We need Lemma 1 and Lemma 2 for Path ORAM [29]. We restate the two lemmas and prove them for RAW Path ORAM.

We adopt the greedy post-processing algorithms G_Z for Path ORAM. We refer the readers to Stefanov et al. [29] for details about G_Z.

Lemma 1. *The Stash usage in a post-processed ∞-ORAM is exactly the same as the Stash usage in Path ORAM after a sequence of RAW ORAM operations:*

$$\mathsf{st}\left(G_Z\left(\mathsf{ORAM}_L^{\infty}\right)\right) = \mathsf{st}\left(\mathsf{ORAM}_L^{Z}\right).$$

To prove this lemma, we made a little change to RAW Path ORAM algorithm. In the original RAW ORAM proposed in this paper, 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 in the similar way to Path ORAM.

Proof. Initially, the lemma obviously holds. We need to show if the lemma holds after m accesses, then after the next ($m+1$-th) access (either RO or RW) it still holds. An RO access adds a block to Stash 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, the lemma still holds. An RW access is exactly the same as an Path ORAM operation, besides that it removes the same set of stale blocks from $\mathsf{ORAM}_L^{Z,A}$ and $\mathsf{ORAM}_L^{\infty,A}$. It is proven that in Path ORAM [29] that after such a Path ORAM access, the lemma still holds. □

Lemma 2. *The Stash usage after post-processing* $\mathsf{st}\left(G_Z\left(\mathsf{ORAM}_L^{\infty}\right)\right) > R$ *if and only if* $\exists T \in \mathsf{ORAM}_L^{\infty}$ *such that* $X(T) > c(T) + R$ *before post-processing.*

The proof for this lemma in Path ORAM still holds for RAW Path ORAM. We refer the readers to Stefanov et al. [29] for details of the proof for Lemma 2 and the induction for RW access in Lemma 1.

By Lemma 1 and Lemma 2, we have

$$\begin{aligned}
&\Pr\left[\mathsf{st}\left(\mathsf{ORAM}_L^{Z,A}\right) > R\right] = \Pr\left[\mathsf{st}\left(G_Z\left(\mathsf{ORAM}_L^{\infty,A}\right)\right) > R\right] \\
&\leq \sum_{T \in \mathsf{ORAM}_L^{\infty,A}} \Pr\left[X(T) > c(T) + R\right] \\
&< \sum_{n \geq 1} 4^n \max_{T:n(T)=n} \Pr\left[X(T) > c(T) + R\right] \qquad (2)
\end{aligned}$$

where $n(T)$ is the total number of nodes in subtree T, $c(T)$ is the maximum number of blocks T can hold (the capacity), and $X(T)$ is the actual number of real blocks that are stored in T.

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* $N \leq A \cdot 2^{L-1}$*, the average load of* T *has the following upper bound:*

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

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

If b is not a leaf bucket, we define two variables M_1 and M_2: the last RW access to b's left child is the M_1-th RW access, and the last RW access to b's right child is the M_2-th RW access. 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 $m \leq M_1$ will not be in b as

they will go to either the left child or the right child of b. Blocks with $m > M_2$ will not be in b as the last access to b (M_2-th) has already occurred. Therefore, only blocks with time stamp $M_1 < m \leq M_2$ can be in b. There are at most $D = A|M_1 - M_2|$ such blocks [6] and each goes to b independently with a probability of $2^{-(i+1)}$, where i is the level of b. The deterministic nature of RW accesses in RAW ORAM makes it easy to find out that $|M_1 - M_2| = 2^i$. Therefore, $E(Y) \leq D \cdot 2^{-(i+1)} = A/2$ for any non-leaf bucket as well. □

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,

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

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,

$$\Pr\left[\mathsf{st}\left(\mathsf{ORAM}_L^{Z,A}\right) > R\right] = \sum_{n \geq 1} \alpha^R \cdot \beta^n < \frac{\alpha^R}{1-\beta}$$

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. Unfortunately, the current proof does not cover the ones actually in use (e.g., Z4A3, Z5A5 §6.1). We leave them to future work.

[6] Only real or stale blocks with the right time stamp will be put in b by the M_2-th access. Some of them may be accessed again after the M_2-th access and become stale. But this does not affect the total number of blocks in b as stale blocks are treated as real blocks.

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 $h \stackrel{?}{=} H-1$ **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 $l_h \stackrel{?}{\sim} = \perp$ **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 $a_e \stackrel{?}{\sim} = \perp$ **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 $op \stackrel{?}{=}$ 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 $op \stackrel{?}{=}$ write **then**
    Stash$[r] \leftarrow (a, l', D')$
  else if $op \stackrel{?}{=}$ read **then**
    Stash$[r] \leftarrow (a, l', D)$
  else if $op \stackrel{?}{=}$ 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 $RAWCnt \stackrel{?}{=} 0$ **then**
    $l_g \leftarrow G \mod 2^L$
    $\mathsf{RWAccess}(l_g)$
    $G \leftarrow G + 1$
  return $\mathsf{ROAccess}(a, l, l', op, D')$
function RWACCESS(l)
  for $i \leftarrow 0$ to L **do** ▷ read path
    bucket $\leftarrow \mathsf{Decrypt}_K(\mathcal{P}(l)[i])$
    $\mathsf{Verify_K}$(bucket)
    $\mathsf{InsertBlocks}(\mathsf{Stash}, \text{bucket})$
  $\mathcal{S} \leftarrow \mathsf{PushToLeaf}(\mathsf{Stash}, l)$
  for $i \leftarrow 0$ to L **do** ▷ write path back
    bucket $\leftarrow \mathcal{S}[i * L, \ldots, i * L + Z - 1]$
    $\mathsf{RemoveBlocks}(\mathsf{Stash}, \text{bucket})$
    $\mathsf{MAC_K}$(bucket)
    $\mathcal{P}(l)[i] \leftarrow \mathsf{Encrypt}_K$(bucket)
function ROACCESS(a, l, l', op, D')
  if $op \stackrel{?}{=}$ append **then**
    $\mathsf{InsertBlocks}(\mathsf{Stash}, (a, l', D'))$ ▷ append and return
    return
  for $i \leftarrow 0$ to L **do** ▷ read path
    header$[i] \leftarrow \mathsf{Decrypt}_K(\mathcal{P}(l)[i].\text{header})$
    ▷ decrypt header
    for $j \leftarrow 0$ to $Z - 1$ **do**
      if header$[i]$.addr$[j] \stackrel{?}{=} a$ **then** ▷ found block a
        bucket $\leftarrow \mathsf{Decrypt}_K(\mathcal{P}(l)[i])$
        $\mathsf{Verify_K}$(bucket)
        $\mathsf{InsertBlocks}(\mathsf{Stash}, \text{bucket}[j])$
        bucket$[j] \leftarrow \perp$
        $\mathsf{MAC_K}$(bucket)
        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
    $\mathcal{P}(l)[i].\text{header} \leftarrow \mathsf{Encrypt}_K(\text{header}[i])$
  return D