

MIT Open Access Articles

Freecursive ORAM: [Nearly] Free Recursion and Integrity Verification for Position-based Oblivious RAM

The MIT Faculty has made this article openly available. **Please share** how this access benefits you. Your story matters.

Citation: Fletcher, Christopher W. "Freecursive ORAM: [Nearly] Free Recursion and Integrity Verification for Position-based Oblivious RAM." Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, 14-18 March, 2015, Istanbul, Turkey, ACM, 2015.

As Published: <http://dx.doi.org/10.1145/2775054.2694353>

Publisher: Association for Computing Machinery (ACM)

Persistent URL: <https://hdl.handle.net/1721.1/121405>

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Terms of use: Creative Commons Attribution-Noncommercial-Share Alike



Freecursive ORAM: [Nearly] Free Recursion and Integrity Verification for Position-based Oblivious RAM

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

[†] Massachusetts Institute of Technology — {cwfletch, renling, kwon, devadas}@mit.edu

[‡] University of Connecticut — vandijk@engr.uconn.edu

Abstract

Oblivious RAM (ORAM) is a cryptographic primitive that hides memory access patterns as seen by untrusted storage. Recently, ORAM has been architected into secure processors. A big challenge for hardware ORAM schemes is how to efficiently manage the Position Map (PosMap), a central component in modern ORAM algorithms. Implemented naïvely, the PosMap causes ORAM to be fundamentally unscalable in terms of on-chip area. On the other hand, a technique called Recursive ORAM fixes the area problem yet significantly increases ORAM’s performance overhead.

To address this challenge, we propose three new mechanisms. We propose a new ORAM structure called the PosMap Lookaside Buffer (PLB) and PosMap compression techniques to reduce the performance overhead from Recursive ORAM empirically (the latter also improves the construction asymptotically). Through simulation, we show that these techniques reduce the memory bandwidth overhead needed to support recursion by 95%, reduce overall ORAM bandwidth by 37% and improve overall SPEC benchmark performance by 1.27 \times . We then show how our PosMap compression techniques further facilitate an extremely efficient integrity verification scheme for ORAM which we call PosMap MAC (PMMAC). For a practical parameterization, PMMAC reduces the amount of hashing needed for integrity checking by $\geq 68\times$ relative to prior schemes and introduces only 7% performance overhead.

We prototype our mechanisms in hardware and report area and clock frequency for a complete ORAM design post-synthesis and post-layout using an ASIC flow in a 32 nm commercial process. With 2 DRAM channels, the design post-layout runs at 1 GHz and has a total area of .47 mm². Depending on PLB-specific parameters, the PLB accounts for 10% to 26% area. PMMAC costs 12% of total design area. Our work is the first to prototype Recursive ORAM or ORAM with any integrity scheme in hardware.

1. Introduction

With cloud computing becoming increasingly popular, privacy of users’ sensitive data has become a large concern in computation outsourcing. In an ideal setting, users would like to “throw their encrypted data over the wall” to a cloud service that should perform computation on that data without the service learning 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 [39] or the encrypted data it is computing upon [16, 21].

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 [9, 11], and there has been significant follow-up work that has resulted in more efficient and cryptographically-secure ORAM schemes [4, 5, 14, 18, 23, 24, 30, 34, 36].

An important use case for ORAM is in *trusted hardware* [7, 21, 26, 31, 34]. In this setting, an on-chip ORAM controller intercepts last-level cache (LLC) misses/evictions and turns them into obfuscated main memory requests (i.e., the address pattern is randomized, data read/written is encrypted). Since program performance is very sensitive to cache miss latency, the ORAM controller logic is implemented directly in hardware.

1.1 Problem: Position Map Management

A big challenge for hardware ORAM controllers is that they need to store and manage a large data structure called the *Position Map* (PosMap for short). Conceptually, the PosMap is a page table that maps data blocks to random locations in external memory. Hence, the PosMap’s size is proportional to the number of data blocks (e.g., cache lines) in main memory and can be hundreds of MegaBytes in size.

PosMap size has been an issue for all prior hardware ORAM proposals. First, Maas et al. [21] recently built Phantom, the first hardware ORAM prototype on an FPGA. The Phantom design stores the whole PosMap on-chip. As a result, to scale beyond 1 GB ORAM capacities, Phantom re-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS ’15, March 14–18, 2015, Istanbul, Turkey.
Copyright © 2015 ACM 978-1-4503-2835-7/15/03...\$15.00.
<http://dx.doi.org/10.1145/2694344.2694353>

quires the use of multiple FPGAs *just to store the PosMap*. Second, Ren et al. [26] evaluate a technique called *Recursive ORAM* [30] in the secure hardware setting. The idea is to store the PosMap in additional ORAMs to reduce the on-chip storage requirement. The cost of Recursive ORAM is performance. One must access all the ORAMs in the recursion on each ORAM access. Even after architectural optimizations [26], a Recursive ORAM spends 39% to 56% of its time looking up PosMap ORAMs, and this percentage only increases with ORAM capacity (§ 3.2.1).

We believe that to be practical and scalable to large ORAM capacities, Recursive ORAM is necessary in secure hardware. To that end, one focus in this paper is to explore novel ways to dramatically reduce the performance overhead of Recursive ORAM.

We then take a new direction and explore what other functions the PosMap can perform on behalf of the processor. We show how our optimized PosMap construction can also, for very little additional cost, be used to perform extremely efficient *integrity verification* for ORAM. Obviously, integrity verification is an important consideration for any secure storage system where data can be tampered with. Yet, prior ORAM integrity schemes based on Merkle trees [25] require large hash unit bandwidth to rate match memory. We show how clever use of our optimized PosMap simplifies this problem dramatically.

1.2 Our Contribution

To accomplish the above goals, we contribute the following mechanisms:

1. We propose the **PosMap Lookaside Buffer** (§ 4), or *PLB* for short, a mechanism that significantly reduces the memory bandwidth overhead of Recursive ORAMs depending on underlying *program address locality*.
2. We propose a way to **compress the PosMap** (§ 5), which reduces the cost of recursion in practice and asymptotically, and improves the PLB’s effectiveness.
3. We then show how to further use PosMap compression to create an ORAM integrity scheme, called **PosMap MAC** (§ 6) or *PMMAC* for short, which is extremely efficient in practice and is asymptotically optimal.

With the PLB and PosMap compression, we reduce PosMap-related memory bandwidth overhead by 95%, reduce overall ORAM bandwidth overhead by 37% and improve SPEC performance by 1.27×. As a standalone scheme, PMMAC reduces the amount of hashing needed for integrity checking by $\geq 68\times$ relative to prior schemes. Using PosMap compression and PMMAC as a combined scheme, we demonstrate an integrity checking mechanism for ORAM that increases performance overhead by only 7%.

In addition to evaluating our proposals using software simulation, we prototype a complete ORAM design with the PLB and PMMAC in a 32 nm commercial process and evaluate the entire design post-synthesis and post-layout for area and clock frequency. Our design is open source and available at <http://kwonalbert.github.io/oram>.

Post-synthesis, PMMAC and the PLB cost $< 13\%$ and 10% (respectively) of total design area. Post-layout, the entire ORAM controller parameterized for 2 DRAM channels is $.47 \text{ mm}^2$ and runs at 1 GHz. Our prototype is the first hardware implementation of Recursive ORAM and integrity verification with ORAM.

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 remote user. 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 access data in 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 and/or learn information about the encrypted data.

Security definition (privacy)

We adopt the following security definition for ORAM that is implicit in all prior hardware proposals [6, 21, 26]:

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

Here, $|\text{ORAM}(\overleftarrow{a})|$ denotes the length of $\text{ORAM}(\overleftarrow{a})$. In other words, the memory request sequence visible to the adversary leaks only its length. Importantly, this definition allows the processor to use conventional on-chip cache. In our terminology, \overleftarrow{a} is the sequence of load/store instructions in the program. $|\text{ORAM}(\overleftarrow{a})|$ is determined by, and thus reveals, the number of LLC misses in \overleftarrow{a} , but not $|\overleftarrow{a}|$. The definition captures the essence of ORAM’s privacy guarantee: ORAM hides individual elements in the data request sequence, while leaking a small amount of information by exposing 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)

We guarantee that ORAM 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 (i.e., is *authentic* and *fresh*).

Security definition (integrity+privacy)

In the presence of active adversaries, we guarantee that ORAM behaves like a valid memory and that the resulting ORAM request sequence is computationally indistinguishable up to the point when tampering is detected by the ORAM controller (i.e., that the ORAM request sequence achieves the privacy guarantees described above).

When tampering is detected, the processor receives an exception at which point it can kill the program, or take some measures to prevent leakage from integrity violation detection time.

Threats outside of scope: timing channel leakage

Following Phantom [21], we assume that each DRAM request made during an ORAM access occurs at data-independent times and thus does not leak information. Also following Phantom (and work in [26]), we do not obfuscate when an ORAM access is made or the time it takes the program to terminate. These two timing channels are orthogonal to our work and have been addressed for ORAM by Fletcher et al. [6]. The schemes in that work can be adopted on top of our work if timing protection is needed.

3. Background

We will evaluate our techniques on top of *Path ORAM* [34], which was the scheme used in [21, 26]. Path ORAM is not only efficient but is also relatively simple algorithmically, making it implementable in hardware. We remark, however, that our optimizations apply to other Position-based (i.e., use a PosMap) ORAM schemes including [8, 30, 32].

3.1 Basic Path ORAM

Hardware Path ORAM is made up of two components: a (trusted) on-chip ORAM controller and (untrusted) external memory.

The untrusted external storage is logically structured as a binary tree, as shown in Figure 1, called the **ORAM tree**. The ORAM tree’s levels range from 0 (the root) to L (the leaves). Each node in the tree is called a *bucket* and has a fixed number of slots (denoted Z) which can store blocks, which are the unit of data requested by the processor (e.g., a cache line). Bucket slots may be empty at any point, and are filled with *dummy blocks*. All blocks in the tree including dummy blocks are encrypted with a probabilistic encryption scheme such as AES counter mode [19] with a randomized session key. Thus, any two blocks (dummy or real) are indistinguishable after encryption. We refer to a path from the root to some leaf l as path l .

The **ORAM controller** is made up of the *position map*, the *stash* and control logic. The position map, PosMap for short, is a lookup table that associates each data block with a random leaf in the ORAM tree. Managing and optimizing the PosMap is the focus of this paper. If N is the maximum number of real data blocks in the ORAM, the PosMap capacity is $N * L$ bits: one mapping per block. The stash is a memory that temporarily stores up to a small number of data blocks (we assume 200 following [26]).

3.1.1 Path ORAM Invariant and Operation

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 on path l or in the stash.* Blocks are stored in the stash or ORAM tree along with their current leaf and block address.

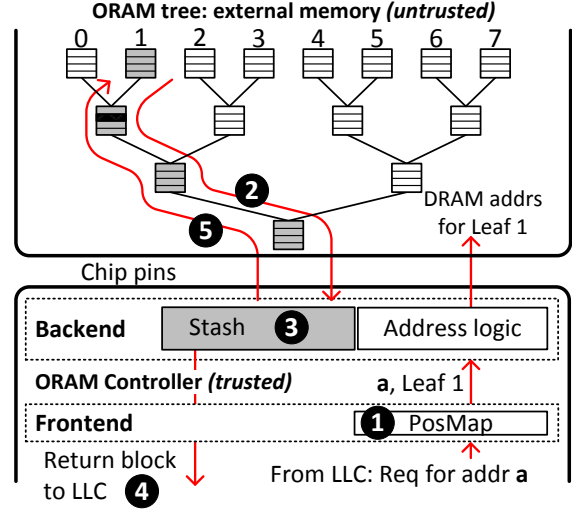


Figure 1. A Path ORAM of $L = 3$ levels and $Z = 4$ slots per bucket. Suppose block a , shaded black, is mapped to path $l = 1$. At any time, block a can be located in any of the shaded structures (i.e., on path 1 or in the stash).

To make a request for a block with address a (block a for short), the LLC calls the ORAM controller via $\text{accessORAM}(a, op, d')$, where op is either read or write and d' is the new data is $op = \text{write}$ (the steps are also shown in Figure 1):

1. Look up PosMap with a , yielding the corresponding leaf label l . Randomly generate a new leaf l' and update the PosMap for a with l' .
2. Read and decrypt all the blocks along path l . Add all the real blocks to the stash (dummies are discarded). Due to the Path ORAM invariant, block a must be in the stash at this point.
3. Update block a in the stash to have leaf l' .
4. If $op = \text{read}$, return block a to the LLC. If $op = \text{write}$, replace the contents of block a with data d' .
5. Evict and encrypt as many blocks as possible from the stash to path l in the ORAM tree (to keep the stash occupancy low) while keeping the invariant. Fill any remaining space on the path with encrypted dummy blocks.

To simplify the presentation, we refer to Step 1 (the PosMap lookup) as the Frontend(a), or Frontend, and Steps 2-5 as the Backend(a, l, l', op, d'), or Backend. This work optimizes the Frontend and the techniques can be applied to any Position-based ORAM Backend.

3.1.2 Path ORAM Security

The intuition for Path ORAM’s security is that every PosMap lookup (Step 1) will yield a fresh random leaf to

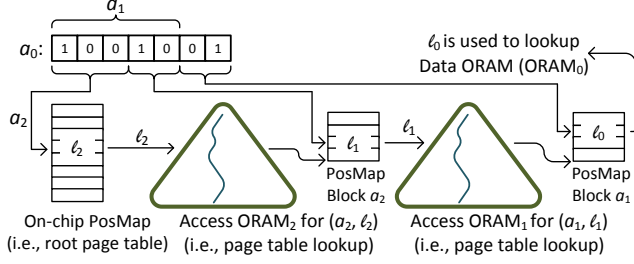


Figure 2. Recursive ORAM with PosMap block sizes $X = 4$, making an access to the data block with program address $a_0 = 1001001_2$. Recursion shrinks the PosMap capacity from $N = 128$ to 8 entries.

access the ORAM tree for that access. This makes the sequence of ORAM tree paths accessed independent of the actual program address trace. Probabilistic encryption hides *which* block is accessed on the path. Further, stash overflow probability is negligible if $Z \geq 4$ (proven for $Z \geq 5$ [34] and shown experimentally for $Z = 4$ [21]).

3.2 Recursive ORAM

As mentioned in § 3.1, the number of entries in the PosMap scales linearly with the number of data blocks in the ORAM. This results in a significant amount of on-chip storage (hundreds of KiloBytes to hundreds of MegaBytes). To address this issue, Shi et al. [30] proposed scheme called *Recursive ORAM*, which has been studied in simulation for trusted hardware proposals [26]. The basic idea is to store the PosMap in a separate ORAM, and store the new ORAM’s (smaller) PosMap on-chip. *We make an important observation that the mechanics of Recursive ORAM are remarkably similar to multi-level page tables in traditional virtual memory systems.* We use this observation to help explain ideas and derive optimizations.

We explain Recursive ORAM through the example in Figure 2, which uses two levels of recursion. The system now contains 3 separate ORAM trees: the *Data ORAM*, denoted as ORam_0 , and two *PosMap ORAMs*, denoted ORam_1 and ORam_2 . Blocks in the PosMap ORAMs are akin to page tables. We say that PosMap blocks in ORam_i store X leaf labels which refer to X blocks in ORam_{i-1} . This is akin to having X pointers to the next level page table and X is a parameter.¹

Suppose the LLC requests block a_0 , stored in ORam_0 . The leaf label l_0 for block a_0 is stored in PosMap block $a_1 = a_0/X$ of ORam_1 (all division is floored). Like a page table, block a_1 stores leaves for neighboring data blocks (i.e., $\{a_0, a_0 + 1, \dots, a_0 + X - 1\}$ in the case where a_0 is a multiple of X). The leaf l_1 for block a_1 is stored in the block $a_2 = a_0/X^2$ stored in ORam_2 . Finally, leaf l_2 for PosMap block a_2 is stored in the on-chip PosMap. The on-chip PosMap is now akin to the root page table, e.g., register CR3 on X86 systems.

¹ Generally, each PosMap level can have a different X . We assume the same X for all PosMaps for simplicity.

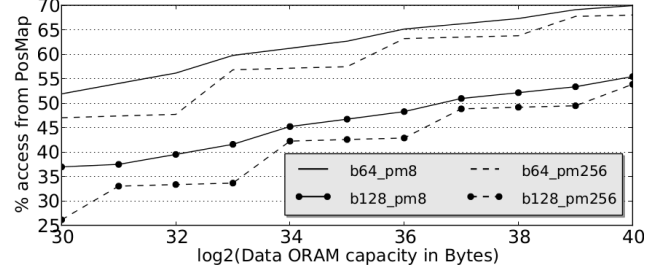


Figure 3. The percentage of Bytes read from PosMap ORAMs in a full Recursive ORAM access for $X = 8$ (following [26]) and $Z = 4$. All bucket sizes are padded to 512 bits to estimate the effect in DDR3 DRAM. The notation b64_pm8 means the ORAM block size is 64 Bytes and the on-chip PosMap is at most 8 KB.

To make a Data ORAM access, we must first lookup the on-chip PosMap, ORam_2 and ORam_1 in that order. Thus, a Recursive ORAM access is akin to a full page table walk. Additional PosMap ORAMs ($\text{ORam}_3, \text{ORam}_4, \dots, \text{ORam}_{H-1}$) may be added as needed to shrink the on-chip PosMap further. H denotes the total number of ORAMs including the Data ORAM in the recursion and $H = \log(N/p)/\log X + 1$ if p is the number of entries in the on-chip PosMap. All logarithms will be base 2 for the rest of the paper.

3.2.1 Overhead of Recursion

It should now be clear that Recursive ORAM increases total ORAM access latency. Unintuitively, with small block sizes, PosMap ORAMs can contribute to more than half of the total ORAM latency as shown in Figure 3. For a 4 GB Data ORAM capacity, 39% and 56% of bandwidth is spent on looking up PosMap ORAMs (depending on block size), and increasing the on-chip PosMap only slightly dampens the effect. Abrupt kinks in the graph indicate when another PosMap ORAM is added (i.e., when H increases).

We now explain this overhead from an asymptotic perspective. We know a single Path ORAM (without recursion) with a block size B of bits transfers $O(B \log N)$ bits per access. In Recursive ORAM, the best strategy to minimize bandwidth is to set X to be constant, resulting in a PosMap ORAM block size of $B_p = \Theta(\log N)$. Then, the number of PosMap ORAMs needed is $\Theta(\log N)$, and the resulting bandwidth overhead becomes

$$O\left(\log N + \frac{HB_p \log N}{B}\right) = O\left(\log N + \frac{\log^3 N}{B}\right)$$

The first term is for Data ORAM and the second term accounts for all PosMap ORAMs combined. In realistic processor settings, $\log N \approx 25$ and data block size $B \approx \log^2 N$ (512 or 1024 in Figure 3). Thus, it is natural that PosMap ORAMs account for roughly half of the bandwidth overhead.

In the next section, we show how insights from traditional virtual memory systems, coupled with security mechanisms, can dramatically reduce this PosMap ORAM overhead (§ 4).

4. PosMap Lookaside Buffer

Given our understanding of Recursive ORAM as a multi-level page table for ORAM (§ 3.2), a natural optimization is to cache PosMap blocks (i.e., page tables) so that LLC accesses exhibiting program address locality require less PosMap ORAM accesses on average. This idea is the essence of the *PosMap Lookaside Buffer*, or PLB, whose name obviously originates from the Translation Lookaside Buffer (TLB) in conventional systems. Unfortunately, unless care is taken, this idea totally breaks the security of ORAM. This section develops the scheme and fixes the security holes.

4.1 High-level Idea and Ingredients

4.1.1 PLB Caches

The key point from § 3.2 is that blocks in PosMap ORAMs contain a set of leaf labels for *consecutive blocks* in the next ORAM. Given this fact, we can eliminate some PosMap ORAM lookups by adding a hardware cache to the ORAM Frontend called the PLB. Suppose the LLC requests block a_0 at some point. Recall from § 3.2 that the PosMap block needed from ORam_i for a_0 has address $a_i = a_0/X^i$. If this PosMap block is in the PLB when block a_0 is requested, the ORAM controller has the leaf needed to lookup ORam_{i-1} , and can skip ORam_i and *all the smaller PosMap ORAMs*. Otherwise, block a_i is retrieved from ORam_i and added to the PLB. When block a_i is added to the PLB, another block may have to be evicted in which case it is appended to the stash of the corresponding ORAM.

A minor but important detail is that a_i may be a valid address for blocks in multiple PosMap ORAMs; to disambiguate blocks in the PLB, block a_i is stored with the tag $i||a_i$ where $||$ denotes bit concatenation.

4.1.2 PLB (In)security

Unfortunately, since each PosMap ORAM is stored in a different physical ORAM tree and PLB hits/misses correlate directly to a program's access pattern, the PosMap ORAM access *sequence* leaks the program's access pattern. To show how this breaks security, consider two example programs in a system with one PosMap ORAM ORam_1 (whose blocks store $X = 4$ leaves) and a Data ORAM ORam_0 . Program A unit strides through memory (e.g., touches $a, a + 1, a + 2, \dots$). Program B scans memory with a stride of X (e.g., touches $a, a + X, a + 2X, \dots$). For simplicity, both programs make the same number of memory accesses. Without the PLB, both programs generate the same access sequence, namely: 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, ... where 0 denotes an access to ORam_0 , and 1 denotes an access to ORam_1 . However, with the PLB, the adversary sees the following access sequences (0 denotes an access to ORam_0 on a PLB hit):

Program A : 1, 0, **0, 0, 0**, 1, 0, **0, 0, 0**, 1, 0, ...

Program B : 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, ...

Program B constantly misses in the PLB and needs to access ORam_1 on every access. Clearly, the adversary can tell program A apart from program B in the PLB-enabled system.

4.1.3 Security Fix: Unified ORAM Tree

To hide PosMap access sequence, we will change Recursive ORAM such that *all PosMap ORAMs and the Data ORAM store blocks in the same physical tree* which we denote ORam_U . This is remotely similar to storing all data in one ORAM “bank” as described in [20], which we compare to further in § 8. Organizationally, the PLB and on-chip PosMap become the new Path ORAM Frontend, which interacts with a single ORAM Backend (§ 3.1). Security-wise, both programs from the previous section access *only* ORam_U with the PLB and the adversary cannot tell them apart (see § 4.3 for more discussion on security).

4.1.4 TLB vs. PLB

We remark that while a traditional TLB caches *single address translations*, the PLB caches entire PosMap blocks (akin to whole page tables). The address locality exploited by both structures, however, is the same.

4.2 Detailed Construction

4.2.1 Blocks Stored in ORam_U

Data blocks and the PosMap blocks originally from the PosMap ORAMs (i.e., $\text{ORam}_1, \dots, \text{ORam}_{H-1}$) are now stored in a single ORAM tree (ORam_U) and all accesses are made to this one ORAM tree. Both data and PosMap blocks now have the same size. Since the number of blocks that used to be stored in some ORam_i ($i > 0$) is X times smaller than the number of blocks stored in ORam_{i-1} , storing PosMap blocks alongside data blocks adds at most one level to ORam_U .

Each set of PosMap blocks must occupy a disjoint address space so that they can be disambiguated. For this purpose we re-apply the addressing scheme introduced in § 4.1.1: Given data block a_0 , the address for the PosMap block originally stored in ORam_i for block a_0 is given by $i||a_i$, where $a_i = a_0/X^i$. This address $i||a_i$ is used to fetch the PosMap block from the ORam_U and to lookup the PosMap block in the PLB. To simplify the notation, we don't show the concatenated address $i||a_i$ in future sections and just call this block a_i .

4.2.2 ORAM readrmv and append Operations

We use two new flavors of ORAM access to support PLB refills/evictions (i.e., *op* in § 3.1): read-remove and append. The idea of these two type of accesses appeared in [26] but we describe them in more detail below. Read-remove (readrmv) is the same as read except that it physically deletes the block in the stash after it is forwarded to the ORAM Frontend. Append (append) adds a block to the stash without performing an ORAM tree access. ORam_U must not contain duplicate blocks: only blocks that are currently not in the ORAM (possibly read-removed previously) can be appended. Further, when a block is appended, the current leaf it is mapped to in ORam_U must be known so that the block can be written back to the ORAM tree during later ORAM accesses.

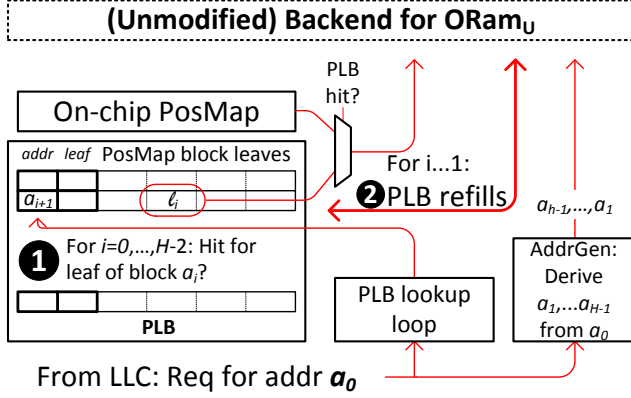


Figure 4. PLB-enabled ORAM Frontend with $X = 4$. Accessing the actual data block a_0 (Step 3 in § 4.2.4) is not shown.

4.2.3 PLB Architecture

The PLB is a conventional hardware cache that stores PosMap blocks. Each PosMap block is tagged with its block address a_i . On a hit, one of the X leaves in the block is read out and remapped. PosMap blocks are read-removed and appended from/to $ORAM_U$. Thus, each block is stored in the PLB alongside its *current leaf*. The PLB itself has normal cache parameters (size, associativity), and we explore how this space impacts performance in § 7.1.3.

4.2.4 ORAM Access Algorithm

The steps to read/write a data block with address a_0 are given below (shown pictorially in Figure 4):

1. **(PLB lookup)** For $i = 0, \dots, H - 2$, look up the PLB for the leaf of block a_i (contained in block a_{i+1}). If one access hits, save i and go to Step 2; else, continue. If no access hits for $i = 0, \dots, H - 2$, look up the on-chip PosMap for the leaf of block a_{H-1} and save $i = H - 1$.
2. **(PosMap block accesses)** While $i \geq 1$, perform a readrmv operation (§ 4.2.2) to $ORAM_U$ for block a_i and add that block to the PLB. If this evicts another PosMap block from the PLB, append that block to the stash. Decrement i . (This loop will not be entered if $i = 0$.)
3. **(Data block access)** Perform an ordinary read or write access to $ORAM_U$ for block a_0 .

Importantly, aside from adding support for readrmv and append, the above algorithm requires no change to the ORAM Backend.

4.3 Security

We now give a proof sketch that our PLB+Unified ORAM tree construction achieves the security definition in § 2. To do this, we use the fact that the PLB interacts with a normal Path ORAM Backend. We make the following observations, which we will use to argue security:

Observation 1. *If all leaf labels l_i used in $\{\text{read}, \text{write}, \text{readrmv}\}$ calls to Backend are random and independent of other l_j for $i \neq j$, the Backend achieves the security of the original Path ORAM (§ 3.1.2).*

Observation 2. *If an append is always preceded by a readrmv, stash overflow probability does not increase (since the net stash occupancy is unchanged after both operations).*

Theorem 1. *The PLB+Unified ORAM tree scheme reduces to the security of the ORAM Backend.*

Proof. The PLB+Unified ORAM Frontend calls Backend in two cases: First, if there is a PLB hit the Backend request is for a PosMap or Data block. In this case, the leaf l sent to Backend was in a PosMap block stored in the PLB. Second, if all PLB lookups miss, the leaf l comes from the on-chip PosMap. In both cases, leaf l was remapped the instant the block was last accessed. We conclude that all $\{\text{read}, \text{write}, \text{readrmv}\}$ commands to Backend are to random/independent leaves and Observation 1 applies. Further, an append command can only be caused by a PLB refill which is the result of a readrmv operation. Thus, Observation 2 applies. \square

Of course, the PLB may influence the ORAM trace length $|ORAM(\overleftarrow{a})|$ by filtering out some calls to Backend for PosMap blocks. Now $|ORAM(\overleftarrow{a})|$ is determined by, and thus reveals, the sum of LLC misses and PLB misses. We remark that processor cache and the PLB are both on-chip and outside the ORAM Backend, so adding a PLB is the same (security-wise) to adding more processor cache: in both cases, only the total number of ORAM accesses leaks. By comparison, using a PLB in without a Unified ORAM tree leaks the *set* of PosMap ORAMs needed on every Recursive ORAM access (§ 4.1.2), which makes leakage grow linearly with $|ORAM(\overleftarrow{a})|$.

5. Compressed PosMap

We now show how to compress the PosMap using pseudorandom functions (PRFs, introduced below). The high level goal is to store *more leaves per PosMap block*, thereby reducing the number of Recursive PosMaps.

5.1 Background: PRFs

A pseudorandom Function, or PRF, family $y = \text{PRF}_K(x)$ is a collection of efficiently-computable functions, where K is a random secret key. A PRF guarantees that anyone who does not know K (even given x) cannot distinguish y from a truly random bit-string in polynomial time with non-negligible probability [10]. For the rest of the paper, we implement $\text{PRF}_K()$ using AES-128.

5.2 Construction

5.2.1 Main Idea

Following previous notation, suppose each PosMap block contains X leaf labels for the next ORAM. For example, some PosMap block contains leaf labels for the blocks with addresses $\{a, a+1, \dots, a+X-1\}$. With the compressed PosMap scheme, the PosMap block's contents are replaced with an α -bit group counter (GC) and X β -bit individual counters (IC):

$$GC \parallel IC_0 \parallel IC_1 \parallel IC_2 \parallel \dots \parallel IC_{X-1}$$

With this format, we can then compute the current leaf label for block $a + j$ through $\text{PRF}_K(a + j || GC || IC_j) \bmod 2^L$. Note that with this technique, the on-chip PosMap is unchanged and still stores an uncompressed leaf per entry.

5.2.2 Block Remap

For $\text{PRF}_K()$ to generate a uniform random sequence of leaves, we must ensure that each $GC || IC_j$ strictly increases (i.e., the $\text{PRF}_K()$ must never see the same input twice). This is achieved by the following modified remapping operation:

When remapping block $a + j$, the ORAM controller first increments its individual counter IC_j . If the individual counter rolls over (becomes zero again), the ORAM controller will increment the group counter GC . This changes the leaf label for all the blocks in the group, so we have to read each block through the Backend, reset its individual counter and remap it to the updated path given by $\text{PRF}_K(a + j || GC + 1 || 0) \bmod 2^L$. In the worst case where the program always requests the same block in a group, we need to reset X individual counters in the group every 2^β accesses.

We remark that this reset operation is very expensive for baseline Recursive ORAM (§ 3.2). In that case, the ORAM controller must make X full Recursive ORAM accesses to reset the individual counters in a certain PosMap ORAM block. Otherwise, it reveals that individual counters have overflowed in that certain ORAM, which is related to the access pattern. On the other hand, using a single Unified ORAM tree as we do to support the PLB (§ 4.1.3) reduces this to X accesses to ORAM_U .

5.2.3 System Impact and the PLB

The compressed PosMap format can be used with or without a PLB and, like the PLB, does not require changes to the Backend. That is, PosMap blocks are stored in their compressed format inside the PLB and ORAM tree/Backend. Uncompressed leaves are generated using the PRF on-demand by the Frontend. Each block stored in the Backend or ORAM tree is still stored alongside its uncompressed leaf label (a one time cost per block), to facilitate ORAM evictions.

5.3 Benefit of Compressed Format (In Practice)

Our scheme *compresses* the PosMap block by setting α , β and X such that $\alpha/X + \beta < L$, implying that the (amortized) bits needed to store each leaf has decreased. A larger X means a fewer number of PosMap ORAMs are needed as discussed in § 3.2. Further, this scheme improves the PLB's hit rate (§ 4.2.4) since more blocks are associated with a given PosMap block.

For concreteness, suppose the ORAM block size in bits is $B = 512$. The compressed PosMap scheme enables $X' = 32$ by setting $\alpha = 64$ and $\beta = 14$, regardless of ORAM tree depth L .² In this configuration, the worst case block remap overhead is $X'/2^\beta = .2\%$ (§ 5.2.2). By comparison,

² We restrict X' to be a power of two to simplify the PosMap block address translation from § 3.2.

the original PosMap representation (up to § 4) only achieves $X = 16$ for ORAM tree depths of $L = 17$ to $L = 32$.

5.4 Benefit of Compressed Format (Theoretical)

We now show that for small data block sizes $B = o(\log^2 N)$, the compressed PosMap with a Unified ORAM asymptotically improves Recursive Path ORAM. In the following analysis, we assume there is no PLB or that the PLB never hits, since there is no good way to model program-dependent locality. We note that by setting $\beta = \log \log N$, and $X' = \log N / \log \log N$, the overhead to reset individual counters is $X'/2^\beta = o(1)$ and we will assume this setting for the rest of the section. Further, as discussed in § 3.2.1 and [33], we always set $B_p = \Theta(\log N)$, because a larger block size for PosMap ORAMs is sub-optimal.

When $B \neq B_p$, we will break each data block into sub-blocks of size B_p , and store them in the Unified ORAM tree as independent blocks. We let these sub-blocks share a single individual counter; the uncompressed leaf for each sub-block is obtained by including the sub-block index k in the PRF input, such that leaves are generated by $\text{PRF}_K(GC || IC_j || a + j || k) \bmod 2^L$. Now a full ORAM access involves H Backend accesses to load the above PosMap block, and another $\lceil B/B_p \rceil$ Backend accesses to load all the data block sub-blocks. The asymptotic bandwidth overhead is

$$(1 + X'/2^\beta) \cdot \frac{2}{B} (B_p \cdot \log N) \cdot \left(\left\lceil \frac{B}{B_p} \right\rceil + H \right) \\ = O \left(\log N + \frac{\log^3 N}{B \log \log N} \right).$$

When $B = o(\log^2 N)$, this result asymptotically outperforms Recursive Path ORAM; When $B = \omega(\log N)$, it beats Kushilevitz et al. [18], the best existing ORAM scheme under small client storage and a small block size. This makes it the best asymptotic ORAM so far for any block size in between.

6. Integrity Verification: PosMap MAC

We now describe a novel and simple integrity verification scheme for ORAM called *PosMap MAC*, or *PMMAC*, that is facilitated by our PosMap compression technique from the previous section. PMMAC achieves asymptotic improvements in hash bandwidth over prior schemes and is easy to implement in hardware.

6.1 Background: MACs

Suppose two parties Alice and Bob share a secret K and Alice wishes to send messages to Bob over an insecure channel where data packets d_i ($i = 0, \dots$) can be tampered by some adversary Eve. To guarantee message *authenticity*, Alice can send Bob tuples (h_i, d_i) where $h_i = \text{MAC}_K(d_i)$ and $\text{MAC}_K()$ is a Message Authentication Code (e.g., a keyed hash function [3]). For the rest of the paper, we implement $\text{MAC}_K()$ using SHA3-224.

The MAC scheme guarantees that Eve can only produce a message forgery (h^*, d^*) with negligible probability, where

$h_\star = \text{MAC}_K(d_\star)$ and d_\star was not transmitted previously by Alice. In other words, without knowing K , Eve cannot come up with a forgery for a message whose MAC it has never seen.

Importantly Eve can still perform a replay attack, violating *freshness*, by replacing some (h_i, d_i) with a previous legitimate message (h_j, d_j) . A common fix for this problem is to embed a *non-repeating counter* in each MAC [29]. Suppose Alice and Bob have shared access to an oracle that, when queried, returns the number of messages sent by Alice but not yet checked by Bob. Then, for message i , Alice transmits (h'_i, d_i) where $h'_i = \text{MAC}_K(i || d_i)$. Eve can no longer replay an old packet (h'_j, d_j) because $\text{MAC}_K(i || d_j) \neq \text{MAC}_K(j || d_j)$ with overwhelming probability. The challenge in implementing these schemes is that Alice and Bob must have access to a shared, *tamper-proof* counter.

6.2 Construction

6.2.1 Main Idea and Non-Recursive PMMAC

Clearly, any memory system including ORAM that requires integrity verification can implement the replay-resistant MAC scheme from the previous section by storing per-block counters in a tamper-proof memory. Unfortunately, the size of this memory is even larger than the original ORAM PosMap making the scheme untenable. *We make a key observation that if PosMap entries are represented as non-repeating counters, as is the case with the compressed PosMap (§ 5.2.1), we can implement the replay-resistant MAC scheme without additional counter storage.*

We first describe PMMAC without recursion and with simple/flat counters per-block to illustrate ideas. Suppose block a which has data d has access count c . Then, the on-chip PosMap entry for block a is c and we generate the leaf l for block a through $l = \text{PRF}_K(a || c) \bmod 2^L$ (i.e., same idea as § 5.2.1). Block a is written to the Backend as the tuple (h, d) where

$$h = \text{MAC}_K(c || a || d)$$

When block a is read, the Backend returns (h_\star, d_\star) and PMMAC performs the following check to verify authenticity/freshness:

$$\text{assert } h_\star == \text{MAC}_K(c || a || d_\star)$$

where \star denotes values that may have been tampered with. After the assertion is checked, c is incremented for the returned block.

Security follows if it is infeasible to tamper with block counters and no counter value for a given block is ever repeated. The first condition is clearly satisfied because the counters are stored on-chip. We can satisfy the second condition by making each counter is wide enough to not overflow (e.g., 64 bits wide).

As with our previous mechanisms, PMMAC requires no change to the ORAM Backend because the MAC is treated

as extra bits appended to the original data block.³ As with PosMap compression, the leaf currently associated with each block in the stash/ORAM tree is stored in its original (uncompressed) format.

6.2.2 Adding Recursion and PosMap Compression

To support recursion, PosMap blocks (including on-chip PosMap entries) may contain either a flat (64 bits) or compressed counter (§ 5.2.1) per next-level PosMap or Data ORAM block. As in the non-Recursive ORAM case, all leaves are generated via a PRF. The intuition for security is that the tamper-proof counters in the on-chip PosMap form the root of trust and then recursively, the PosMap blocks become the root of trust for the next level PosMap or Data ORAM blocks. Note that in the compressed scheme (§ 5.3), the α and β components of each counter are already sized so that each block's count never repeats/overflows. We give a formal analysis for security with Recursive ORAM in § 6.5.

For realistic parameters, the scheme that uses flat counters in PosMap blocks incurs additional levels of recursion. For example, using $B = 512$ and 64 bit counters we have $X = B/64 = 8$. *Importantly, with the compressed PosMap scheme we can derive each block counter from GC and IC_j (§ 5.2.1) without adding levels of recursion or extra counter storage.*

6.3 Key Advantage: Hash Bandwidth and Parallelism

Combined with PosMap compression, the overheads for PMMAC are the bits added to each block to store MACs and the cost to perform cryptographic hashes on blocks. The extra bits per block are relatively low-overhead — the ORAM block size is usually 64-128 Bytes (§ 7.1.5) and each MAC may be 80-128 bits depending on the security parameter.

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. Merkle tree constructions [2, 25] need to integrity verify all the blocks on the path to check/update the root hash. Crucially, our PMMAC construction only needs to integrity verify (check and update) 1 block — namely the block of interest — per access, achieving an asymptotic reduction in hash bandwidth.

To give some concrete numbers, assume $Z = 4$ block slots per ORAM tree bucket following [21, 34]. Then there are $Z * (L + 1)$ blocks per path in ORAM tree, and our construction reduces hash bandwidth by $68\times$ for $L = 16$ and by $132\times$ for $L = 32$. We did not include the cost of reading sibling hashes for the Merkle tree for simplicity.

Integrity verifying only a single block also prevents a serialization bottleneck present in Merkle tree schemes. Consider the scheme from [25], a scheme optimized for Path ORAM. Each hash in the Merkle tree node must be recomputed based on the contents of the corresponding ORAM tree bucket and *its child hashes*, and is therefore fundamentally sequential. If this process cannot keep up with memory bandwidth, it will be the system's performance bottleneck.

³That is, the MAC is encrypted along with the block when it is written to the ORAM tree.

6.4 Adding Encryption: Subtle Attacks and Defenses

Up to this point we have discussed PMMAC in the context of providing integrity only. ORAM must also apply a probabilistic encryption scheme (we assume AES counter mode as done in [26]) to all data stored in the ORAM tree. In this section we first show how the encryption scheme of [26] breaks under active adversaries because the adversary is able to *replay the one-time pads used for encryption*. We show how PMMAC doesn't prevent this attack by default and then provide a fix that applies to PMMAC.

We first show the scheme used by [26] for reference: Each bucket in the ORAM tree contains, in addition to Z encrypted blocks, a seed used for encryption (the *BucketSeed*) that is stored in plaintext. (*BucketSeed* is synonymous to the “counter” in AES counter mode.) If the Backend reads some bucket (Step 2 in § 3.1) whose seed is *BucketSeed*, the bucket will be re-encrypted and written back to the ORAM tree using the one-time pad (OTP) $\text{AES}_K(\text{BucketID} \parallel \text{BucketSeed} + 1 \parallel i)$, where i is the current chunk of the bucket being encrypted.

The above encryption scheme breaks privacy under PMMAC because PMMAC doesn't integrity verify BucketSeed. For a bucket currently encrypted with the pad $P = \text{AES}_K(\text{BucketID} \parallel \text{BucketSeed} \parallel i)$, suppose the adversary replaces the plaintext bucket seed to *BucketSeed* – 1. This modification will cause the contents of that bucket to decrypt to garbage, *but won't trigger an integrity violation under PMMAC unless bucket BucketID contains the block of interest for the current access.* If an integrity violation is not triggered, due to the replay of *BucketSeed*, that bucket will next be encrypted using the *same one-time pad P* again.

Replaying one-time pads obviously causes security problems. If a bucket re-encrypted with the same pad P contains plaintext data D at some point and D' at another point, the adversary learns $D \oplus D'$. If D is known to the adversary, the adversary immediately learns D' (i.e., the plaintext contents of the bucket).

The fix for this problem is relatively simple: To encrypt chunk i of a bucket about to be written to DRAM, we will use the pad $\text{AES}_K(\text{GlobalSeed} \parallel i)$, where *GlobalSeed* is now a single monotonically increasing counter stored in the ORAM controller in a dedicated register (this is similar to the global counter scheme in [27]). When a bucket is encrypted, the current *GlobalSeed* is written out alongside the bucket as before and *GlobalSeed* (in the ORAM controller) is incremented. Now it's easy to see that each bucket will always be encrypted with a fresh OTP which defeats the above attack.

6.5 Security

We now give an analysis for our complete scheme's integrity and privacy guarantees.

6.5.1 Integrity

We show that breaking our integrity verification scheme is as hard as breaking the underlying MAC, and thus attains

the integrity definition from § 2. First, we have the following observation:

Observation 3. *If the first $k - 1$ address and counter pairs (a_i, c_i) 's the Frontend receives have not been tampered with, then the Frontend seeds a MAC using a unique (a_k, c_k) , i.e., $(a_i, c_i) \neq (a_k, c_k)$ for $1 \leq i < k$. This further implies $(a_i, c_i) \neq (a_j, c_j)$ for all $1 \leq i < j \leq k$.*

This property can be seen directly from the algorithm description, with or without the PLB and/or PosMap compression. For every a , we have a dedicated counter, sourced from the on-chip PosMap or the PLB, that increments on each access. If we use PosMap compression, each block counter will either increment (on a normal access) or jump to the next multiple of the group counter in the event of a group remap operation (§ 5.2.2). Thus, each address and counter pair will be different from previous ones. We now use Observation 3 to prove the security of our integrity scheme.

Theorem 2. *Breaking the PMMAC scheme 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 uses (a_1, c_1) , to call Backend for (h_1, d_1) where $h_1 = \text{MAC}_K(c_1 \parallel a_1 \parallel d_1)$. (a_1, c_1) is unique since there are no previous (a_i, c_i) 's. Note that a_1 and c_1 cannot be tampered with since they come from the Frontend. Thus producing a forgery (h'_1, d'_1) where $d'_1 \neq d_1$ and $h'_1 = \text{MAC}_K(c_1 \parallel a_1 \parallel d'_1)$ is as hard as breaking the underlying MAC. Suppose no integrity violation has happened and Theorem 2 holds up to access $n - 1$. Then the Frontend sees fresh and authentic (a_i, c_i) 's for $1 \leq i \leq n - 1$. By Observation 3, (a_n, c_n) will be unique and $(a_i, c_i) \neq (a_j, c_j)$ for all $1 \leq i < j \leq n$. This means the adversary cannot perform a replay attack (§ 6.1) because all (a_i, c_i) 's are distinct from each other and are tamper-proof. It is also hard to generate a valid MAC with unauthentic data without the secret key. Being able to produce a forgery (h'_i, d'_i) where $d'_i \neq d_i$ and $h'_i = \text{MAC}_K(c_i \parallel a_i \parallel d'_i)$ means the adversary can break the underlying MAC. \square

6.5.2 Privacy

The system's privacy guarantees require certain assumptions under PMMAC because PMMAC is an *authenticate-then-encrypt* scheme [17]. Since the integrity verifier only checks the block of interest returned to the Frontend, other (tampered) data on the ORAM tree path will be written to the stash and later be written back to the ORAM tree. For example, if the adversary tampers with the block-of-interest's address bits, the Backend won't recognize the block and won't be able to send any data to the integrity verifier (clearly an error). The adversary may also coerce a stash overflow by replacing dummy blocks with real blocks or duplicate blocks along a path.

To address these cases, we have to make certain assumptions about how the Backend will possibly behave in the presence of tampered data. We require a correct implementation of the ORAM Backend to have the following property:

Property 1. *If the Backend makes an ORAM access, it only reveals to the adversary (a) the leaf sent by the Frontend for that access and (b) a fixed amount of encrypted data to be written back to the ORAM tree.*

If Property 1 is satisfied, it is straightforward to see that any memory request address trace generated by the Backend is indistinguishable from other traces of the same length. That is, the Frontend receives tamper-proof responses (by Theorem 2) and therefore produces independent and random leaves. Further, the global seed scheme in § 6.4 trivially guarantees that the data written back to memory gets a fresh pad.

If Property 1 is satisfied, the system can still leak the ORAM request trace *length*; i.e., when an integrity violation is detected, or when the Backend enters an illegal state. Conceptually, an integrity violation generates an exception that can be handled by the processor. When that exception is generated and how it is handled can leak some privacy. For example, depending on how the adversary tampered with memory, the violation may be detected immediately or after some period of time depending on whether the tampered bits were of interest to the Frontend. Quantifying this leakage is outside our scope, but we remark that this level of security matches our combined privacy+integrity definition from § 2.

7. Evaluation

We now evaluate our proposals in simulation and with a complete hardware prototype.

7.1 Software Simulation

7.1.1 Methodology and Parameters

We first evaluate our proposals using the Graphite simulator [22] with the processor parameters listed in Table 1. The core and cache model remain the same in all experiments; unless otherwise stated, we assume the ORAM parameters from the table. We use a subset of SPEC06-int benchmarks [15] with reference inputs. All workloads are warmed up over 1 billion instructions and then run for 3 billion instructions.

We derive AES/SHA3 latency, Frontend and Backend latency directly from our hardware prototype in § 7.2. Frontend latency is the time to evict and refill a block from the PLB (§ 4.2.4) and occurs at most once per Backend call. Backend latency (approximately) accounts for the cycles lost due to hardware effects such as serializers/buffer latency/etc and is added on top the time it takes to read/write an ORAM tree path in DRAM, which is given in § 7.1.2.

We model DRAM and ORAM accesses on top of commodity DRAM using DRAMSim2 [28] and its default DDR3_micron configuration with 8 banks, 16384 rows and 1024 columns per row. Each DRAM channels runs at 667 MHz DDR with a 64-bit bus width and provides ~ 10.67 GB/s peak bandwidth. All ORAM configurations assume 50% DRAM utilization (meaning a 4 GB ORAM requires 8 GB of DRAM) and use the subtree layout scheme from [26] to achieve nearly peak DRAM bandwidth.

Table 1. Processor Configuration.

Core, on-chip cache and DRAM	
core model	in order, single issue, 1.3 GHz
add/sub/mul/div	1/1/3/18 cycles
fadd/fsub/fmul/fdiv	3/3/5/6 cycles
L1 I/D cache	32 KB, 4-way, LRU
L1 data + tag access time	1 + 1 cycles
L2 Cache	1 MB, 16-way, LRU
L2 data + tag access time	8 + 3 cycles
cache line size	64 B
Path ORAM/ORAM controller	
ORAM controller clock frequency	1.26 GHz
data block size	64 B
data ORAM capacity	4 GB ($N = 2^{26}$)
block slots per bucket (Z)	4
AES-128 latency	21 cycles (§ 7.2)
SHA3-224 latency (PMMAC)	18 cycles (§ 7.2)
Frontend latency	20 cycles (§ 7.2)
Backend latency	30 cycles (§ 7.2)
Memory controller and DRAM	
DRAM channels	2 (~ 21.3 GB peak bandwidth)
DRAM latency	given by DRAMSim2 [28]

7.1.2 ORAM Latency and DRAM Channel Scalability

ORAM latency is sensitive to DRAM bandwidth and for this reason we explore how changing the channel count impacts ORAM access time in Table 2. ORAM Tree latency refers to the time needed for the Backend to read/write a path in the Unified ORAM tree, given the ORAM parameters in Table 1. All latencies are in terms of *processor* clock cycles, and represent an average over multiple accesses. For reference, a DRAM access for an insecure system without ORAM takes on average 58 processor cycles.

Table 2. ORAM access latency by DRAM channel count.

DRAM channel count	1	2	4	8
ORAM Tree latency (cycles)	2147	1208	697	463

Generally, ORAM latency decreases with channel count as expected but the effect becomes increasingly sub-linear for larger channel counts due to DRAM channel conflicts. Since 2 channels represent realistic mid-range systems and do not suffer significantly from this problem, we will use that setting for the rest of the evaluation unless otherwise specified.

7.1.3 PLB Design Space

Figure 5 shows how direct-mapped PLB capacity impacts performance. For a majority of benchmarks, larger PLBs add small benefits ($\leq 10\%$ improvements). The exceptions are bzip2 and mcf, where increasing the PLB capacity from 8 KB to 128 KB provides 67% and 49% improvement, respectively. We tried increasing PLB associativity (not shown for space) and found that, with a fixed PLB capacity, a fully associative PLB improves performance by $\leq 10\%$ when compared to direct-mapped. To keep the architecture simple, we therefore assume direct-mapped PLBs from now on. Going from a 64 KB to 128 KB direct-mapped PLB, average performance only increases by only 2.7%, so we assume a 64 KB direct-mapped PLB for the rest of the evaluation.

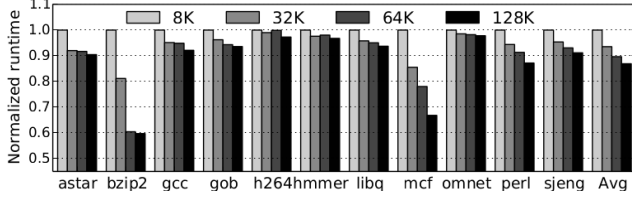


Figure 5. PLB design space, sweeping direct-mapped PLB capacity. Runtime is normalized to the 8 KB PLB point.

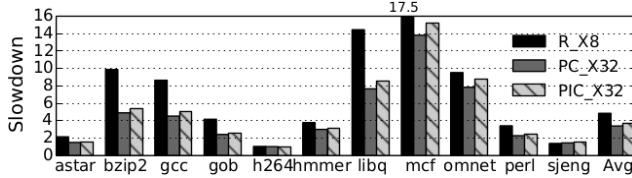


Figure 6. Performance of PLB, Compressed PosMap and PM-MAC. Slowdown is relative to an insecure system without ORAM.

7.1.4 Scheme Composability

We now present our main result (Figure 6), the impact on performance when we compose PLB (§ 4), PosMap compression (§ 5) and PMMAC (§ 6). To name our schemes in the discussion, we use the letters P, I and C to indicate the **PLB**, **I**ntegrity verification (PMMAC) and **C**ompressed PosMap, respectively. For example, PC_X32 denotes PLB+Compressed PosMap with $X = 32$. PI_X8 is the flat-counter PMMAC scheme from § 6.2.2. For PC_X32 and PIC_X32, we apply recursion until the on-chip PosMap is ≤ 128 KB in size, yielding 4 KB on-chip PosMaps for both points. R_X8 is a Recursive ORAM baseline with $X = 8$ (32-Byte PosMap ORAM blocks following [26]) and $H = 4$, giving it a 272 KB on-chip PosMap.

Despite consuming less on-chip area, PC_X32 achieves a $1.43\times$ speedup (30% reduction in execution time) over R_X8 (geomean). To provide integrity, PIC_X32 only adds 7% overhead on top of PC_X32, which is due to the extra bandwidth needed to transfer per-block MACs (§ 6.3).

To give more insight, Figure 7 shows the average data movement per ORAM access (i.e., per LLC miss+eviction). We give the Recursive ORAM R_X8 up to a 256 KB on-chip PosMap. As ORAM capacity increases, the overhead from accessing PosMap ORAMs grows quickly for R_X8. All schemes using a PLB have much better scalability. For the 4 GB ORAM, on average, PC_X32 reduces PosMap bandwidth overhead by 82% and overall ORAM bandwidth overhead by 38% compared with R_X8. At the 64 GB capacity, the reduction becomes 90% and 57%. Notably the PM-MAC scheme without compression (PI_X8) causes nearly half the bandwidth to be PosMap related, due to the large counter width and small X (§ 6.2.2). Compressed PosMap (PIC_X32) solves this problem.

7.1.5 Comparison to Prior-art Recursive ORAM ([26])

While we have applied the ideas from [26] to our baseline in Figure 6, we do change some processor/DRAM parameters

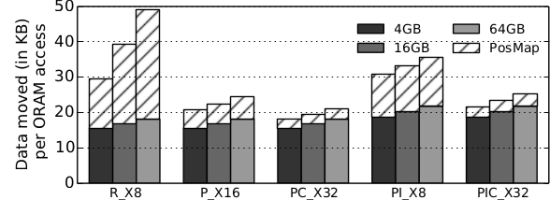


Figure 7. Scalability to large ORAM capacities. White shaded regions indicate data movement from PosMap ORAM lookups. Slowdown is relative to an insecure system without ORAM.

from that work to more realistic values or to be compatible with existing processors, including DRAM bandwidth, processor frequency and cache line size.

For a more apples-to-apples comparison with [26], we now adopt all the parameters from that work: 4 DRAM channels, a 2.6 GHz processor, a 128-Byte cache line and ORAM block size, $Z = 3$, etc. PC_X64 is a PLB-enabled ORAM with a 128-Byte block (cache line) size (thus X doubles). PC_X64 reduces PosMap traffic by 95% and overall ORAM traffic by 37% over the Recursive ORAM configuration from [26].

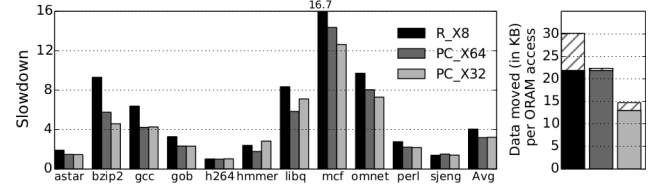


Figure 8. (Left) Performance of Recursive ORAM baseline and our scheme using all parameters from [26]. Slowdown is relative to an insecure system without ORAM. (Right) Average data movement per access. White shaded regions indicate data movement from PosMap ORAM lookups.

PC_X32 is a PLB-enabled ORAM with a 64-Byte block (cache line) size. This configuration has a smaller ORAM latency but also fetches less data per ORAM access. As shown in Figure 8, both PC_X64 and PC_X32 achieve about $1.27\times$ speedup over Recursive ORAM baseline. This shows our scheme is effective for both ORAM block sizes. The larger ORAM block size of PC_X64 benefits benchmarks with good locality (hmmer, libq) but hurts those with poor locality (bzip, mcf, omnetpp).

7.1.6 Comparison to Non-Recursive ORAM with Large Blocks ([21])

In Figure 9, we compare our proposal to the parameterization used by Phantom [21]. Phantom was evaluated with a large ORAM block size (4 KB) so that the on-chip PosMap could be contained on several FPGAs without recursion.

To match § 7.1.4, we model the Phantom parameters on 2 DRAM channels (which matches the DRAM bandwidth reported in [21]) and with a 4 GB ORAM ($N = 2^{20}$, $L = 19$) $Z = 4$, 4 KB blocks and no recursion. For these parameters, the on-chip PosMap is ~ 2.5 MB (which we evaluate for on-chip area in § 7.2.3). To accurately reproduce Phan-

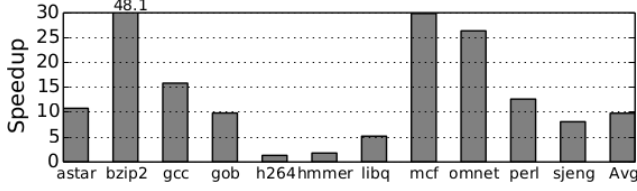


Figure 9. PC_X32 speedup relative to Phantom [21] w/ 4 KB blocks.

Table 3. ORAM area breakdown post-synthesis.

		DRAM channels (nchannel)		
		1	2	4
Area (% of total)	Frontend	31.2	30.0	22.5
	PosMap	7.3	7.0	5.3
	PLB	10.2	9.7	7.3
	PMMAC	12.4	11.9	8.8
	Misc	1.3	1.4	1.1
	Backend	68.8	70.0	77.5
	Stash	28.3	28.9	21.9
	AES	40.5	41.1	55.6
Total cell area (mm ²)		.316	.326	.438

tom’s system performance, we implemented the Phantom block buffer (Section 5.7 of that work) as a 32 KB memory with the CLOCK eviction strategy and assume the Phantom processor’s cache line size is 128 Bytes (as done in [21]).

On average, PC_X32 from § 7.1.4 achieves 10× speedup over the Phantom configuration with 4 KB blocks. The intuition for this result is that Byte movement per ORAM access for our scheme is roughly $(26 \times 64) / (19 \times 4096) = 2.1\%$ that of Phantom. While PC_X32 needs to access PosMap blocks due to recursion, this effect is outweighed by the reduction in Data ORAM Byte movement.

7.2 Hardware Prototype

We now evaluate a complete hardware ORAM prototype pushed through Synopsis’ ASIC synthesis tool Design Compiler and Place and Route tool IC Compiler. Our design (Verilog, tests, etc) is open source and can be found at <http://kwonAlbert.github.io/oram>.

Our first objective is to show that the PLB (§ 4) and PMMAC (§ 6) mechanisms impose a small area overhead. We have not built the compressed PosMap (§ 5) in hardware, but do not think the additional control logic will have a significant effect on area or clock frequency. (Thus, our prototype is equivalent to PI_X8 from § 7.1.4.) More generally, our work is the first to prototype any ORAM through an ASIC hardware flow, and the first to build Recursive ORAM or ORAM with integrity verification in any form of hardware.

7.2.1 Methodology

Our design is written in plain Verilog and was synthesized using a 32 nm commercial standard cell library and memory (SRAM and register file) generator. Our Frontend (the focus of this paper) contains the PLB and PMMAC and has a 64-bit/cycle data interface with the last-level cache for all experiments. ORAM parameters follow Table 1 except that

we use an 8 KB PosMap and 8 KB direct-mapped PLB by default (we discuss a 64 KB PLB design in § 7.2.3). Our Backend is similar to the Phantom Backend [21] and has a $64 \times \text{nchannel}$ bit/cycle datapath to/from DRAM.⁴

To implement cryptographic operations, we used two AES cores from OpenCores [1]: a 21-cycle pipelined core to implement the Backend’s read/write path decryptions/encryptions (§ 3.1) and a non-pipelined 12 cycle core to implement $\text{PRF}_K()$ (§ 5.1). We used a SHA3-224 core from OpenCores to implement $\text{MAC}_K()$ for PMMAC (§ 6.1).

For both synthesis and place and route, we built the design hierarchically as three main components: the Frontend, the stash (Backend) and the AES units used to decrypt/re-encrypt paths (Backend). For synthesis results, we report total cell area; i.e., the minimum area required to implement the design. For layout results, we set a bounding box for each major block that maximized utilization of available space while meeting timing requirements. The entire design required 5 SRAM/RF memories (which we manually placed during layout): the PLB data array, PLB tag array, on-chip PosMap, stash data array and stash tag array. Numerous other (small) buffers were needed and implemented in standard cells.

7.2.2 Results

Post-Synthesis. Table 3 shows ORAM area across several DRAM channel (nchannel) counts. All three configurations met timing using up to a 1.3 GHz clock with a 100 ps uncertainty. The main observation is that the Frontend constitutes a minority of the total area and that this percentage decreases with nchannel. Area decreases with nchannel because the Frontend performs a very small DRAM bandwidth-independent amount of work (~ 50 clock cycles including integrity verification) per Backend access. The Backend’s bits/cycle throughput (AES, stash read/write, etc), on the other hand, must rate match DRAM. Thus, the area cost of the Frontend (and our PMMAC and PLB schemes) is effectively amortized with nchannel (Table 3).⁵

Post-Layout. We now report post-layout area for the nchannel = 2 configuration to give as close an estimate for tape-out as possible. Between synthesis and layout, area increased per-block for nchannel = 2 as follows: Frontend grew by 38%, the stash by 24% and AES by 63%. This configuration met timing at 1 GHz and had a final area of .47 mm². In all cases, some space was sacrificed but the overall picture remains the same.

7.2.3 Alternative Designs

To give additional insight, we estimate the area overhead of *not* supporting recursion or having a larger PLB with

⁴ Our design, however, does not have a DRAM buffer (see [21]). We remark that if such a structure is needed it should be much smaller than that in Phantom (< 10 KB as opposed to hundreds of KiloBytes) due to our 64 Byte block size.

⁵ We note the following design artifact: since we use AES-128, area increases only slightly from nchannel = 1 to 2, because both 64-bit and 128-bit datapaths require the same number of AES units.

recursion. For a 4 GB ORAM, the PosMap must contain 2^{26} to 2^{20} entries (for block sizes 64 Bytes to 4 KB). With a 2^{20} -entry on-chip PosMap, without Recursion, the $n_{\text{channel}} = 2$ design requires $\sim 5 \text{ mm}^2$ area — an increase of over $10\times$. Doubling the ORAM capacity (roughly) doubles this cost. Further, for performance reasons, we prefer smaller block sizes which exacerbates the area problem (§ 7.1.6). On the other hand, we calculate that using Recursion with a 64 KB PLB (to match experiments in § 7.1) increases the area for the $n_{\text{channel}} = 1$ configuration by 29% (and is 26% of total area).

8. Related Work

Numerous prior works [8, 9, 11–14, 18, 32, 34, 36] have significantly improved the theoretical performance of ORAM over the past three decades. Notably among them, Path ORAM [34] is conceptually simple and the most efficient under small client (on-chip) storage. For these reasons, it was embraced by trusted hardware proposals including our work.

Phantom [21] is the first hardware implementation of non-Recursive Path ORAM. In our terminology, Phantom primarily optimizes the ORAM Backend and is therefore complementary to our work. Our Frontend optimizations (the PLB, PMMAC, and compressed PosMap) can be integrated with the Phantom Backend and this is similar to the design we evaluate in § 7.2. Ren et al. [26] explored the Recursive Path ORAM design space through simulation and proposed several optimizations to Recursive Path ORAM. We use their optimized proposal as a baseline in our work. Ascend [7, 38] is a holistic secure processor proposal that uses ORAM for memory obfuscation and adds timing protection on top of ORAM [6]. We believe our techniques can apply to that work as well.

Liu et al. [20] define memory trace obliviousness which can apply to secure processor settings like this paper. That work allocates different program variables into different ORAM trees to increase performance, when the ORAM tree access sequence can be proven to not leak secrets. Similarly, we store PosMap blocks in the same tree because PosMap accesses *do* leak secrets (§ 4.1). Our complete scheme, the PLB plus Unified ORAM tree, however, is completely counter-intuitive given Liu et al.: we show how using a *single* ORAM tree can unlock great performance *improvement*.

Wang et al. [35] develop mechanisms to reduce the overhead of Recursive ORAM, as does our PLB construction. 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). The PLB can be used given any program access pattern.

Splitting PosMap block entries into two counters (which we do for PosMap compression, § 5.2.1) is similar to the split counter scheme due to Yan et al [37], although the motivation in that work is to save space for encryption initialization vectors. Additionally, that work does not hide *when* individual counters overflow, which we require for privacy and describe in § 5.2.2.

9. Conclusion

In this paper we presented three techniques to manage and optimize the Oblivious RAM Position Map (PosMap). We introduced the PosMap Lookaside Buffer (PLB) to decrease the performance overhead of recursion. We then presented a technique to compress the PosMap, which improves the PLB and improves the Recursive Path ORAM construction asymptotically and in practice. Finally, facilitated by PosMap compression, we propose PosMap MAC (PMMAC) to get integrity verification for $\geq 68\times$ less hashing than prior schemes.

Our simulation results show our techniques decrease PosMap (overall) ORAM performance overhead by 95% (37%), which translates to a $1.27\times$ speedup for SPEC workloads. Using PosMap compression, PMMAC degrades performance by only 7%. Our hardware prototype shows how integrity verification costs $\leq 13\%$ of total design area across various DRAM bandwidths.

Acknowledgments: We thank Charles Herder for helpful suggestions and for reviewing early versions of this paper. We also thank the anonymous reviewers for many constructive comments. This research was partially supported by QCRI under the QCRI-CSAIL partnership and by the National Science Foundation. Christopher Fletcher was supported by a DoD National Defense Science and Engineering Graduate Fellowship.

References

- [1] Open cores. <http://opencores.org/>.
- [2] D. Apon, J. Katz, E. Shi, and A. Thiruvengadam. Verifiable oblivious storage. In *PKC*, 2014.
- [3] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *CRYPTO*, 1996.
- [4] 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.
- [5] I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious RAM without random oracles. In *TCC*, 2011.
- [6] 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 *HPCA*, 2014.
- [7] C. Fletcher, M. van Dijk, and S. Devadas. Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *STC*, 2012.
- [8] 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 *PET*, 2013.
- [9] O. Goldreich. Towards a theory of software protection and simulation on oblivious rams. In *STOC*, 1987.
- [10] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM*, 1986.
- [11] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. In *Journal of the ACM*, 1996.

- [12] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *CCSW*, New York, NY, 2011.
- [13] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Practical oblivious storage. In *CODASPY*, New York, NY, 2012.
- [14] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, 2012.
- [15] J. L. Henning. Spec cpu2006 benchmark descriptions. *Computer Architecture News*, 2006.
- [16] M. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.
- [17] H. Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is ssl?). In *CRYPTO*, 2001.
- [18] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in) security of hash-based oblivious ram and a new balancing scheme. In *SODA*, 2012.
- [19] H. Lipmaa, P. Rogaway, and D. Wagner. Comments to NIST concerning AES-modes of operations: CTR-mode encryption. In *Symmetric Key Block Cipher Modes of Operation Workshop*, 2000.
- [20] C. Liu, M. Hicks, and E. Shi. Memory trace oblivious program execution. In *CSF*, 2013.
- [21] 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. In *CCS*, 2013.
- [22] J. E. Miller, H. Kasture, G. Kurian, C. G. III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. In *HPCA*, 2010.
- [23] R. Ostrovsky. Efficient computation on oblivious rams. In *STOC*, 1990.
- [24] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *STOC*, 1997.
- [25] L. Ren, C. Fletcher, X. Yu, M. van Dijk, and S. Devadas. Integrity verification for path oblivious-ram. In *HPCA*, 2013.
- [26] 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 *ISCA*, 2013.
- [27] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin. Using address independent seed encryption and bonsai merkle trees to make secure processors os- and performance-friendly. In *MICRO*, 2007.
- [28] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, 2011.
- [29] L. F. G. Sarmata, 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 *STC*, 2006.
- [30] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious ram with $o((\log n)^3)$ worst-case cost. In *Asiacrypt*, 2011.
- [31] E. Stefanov and E. Shi. Oblivstore: High performance oblivious cloud storage. In *S&P*, 2013.
- [32] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *NDSS*, 2012.
- [33] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. volume abs/1202.5150, 2012.
- [34] 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 *CCS*, 2013.
- [35] X. Wang, K. Nayak, C. Liu, E. Shi, E. Stefanov, and Y. Huang. Oblivious data structures. *IACR*, 2014.
- [36] P. Williams and R. Sion. Single round access privacy on outsourced storage. In *CCS*, 2012.
- [37] C. Yan, D. Engländer, M. Prvulovic, B. Rogers, and Y. Solihin. Improving cost, performance, and security of memory encryption and authentication. *Computer Architecture News*, 2006.
- [38] 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 *CCSW*, 2013.
- [39] X. Zhuang, T. Zhang, and S. Pande. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. In *ASPLOS*, 2004.