ρ : Relaxed Hierarchical ORAM

Chandrasekhar Nagarajan
University of Utah
Salt Lake City, Utah
chandrunaga94@gmail.com

Ali Shafiee
University of Utah
Salt Lake City, Utah
shafiee@cs.utah.edu

Rajeev Balasubramonian
University of Utah
Salt Lake City, Utah
rajeev@cs.utah.edu

Mohit Tiwari
University of Texas, Austin
Austin, Texas
tiwari@austin.utexas.edu

Abstract

Applications in the cloud are vulnerable to several attack scenarios. In one possibility, an untrusted cloud operator can examine addresses on the memory bus and use this information leak to violate privacy guarantees, even if data is encrypted. The Oblivious RAM (ORAM) construct was introduced to eliminate such information leak and these frameworks have seen many innovations in recent years. In spite of these innovations, the overhead associated with ORAM is very significant.

This paper takes a step forward in reducing ORAM memory bandwidth overheads. We make the case that, similar to a cache hierarchy, a lightweight ORAM that fronts the full-fledged ORAM provides a boost in efficiency. The lightweight ORAM has a smaller capacity and smaller depth, and it can relax some of the many constraints imposed on the full-fledged ORAM. This yields a 2-level hierarchy with a relaxed ORAM and a full ORAM. The relaxed ORAM adopts design parameters that are optimized for efficiency and not capacity. We introduce a novel metadata management technique to further reduce the bandwidth for relaxed ORAM access. Relaxed ORAM accesses preserve the indistinguishability property and are equipped with an integrity verification system. Finally, to eliminate information leakage through LLC and relaxed ORAM hit rates, we introduce a deterministic memory scheduling policy. On a suite of memory-intensive applications, we show that the best Relaxed Hierarchical ORAM (ρ) model yields a performance improvement of 50%, relative to a Freecursive ORAM baseline.

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304045

CCS Concepts • **Security and privacy → Hardware-based security protocols**; • **Computer systems organization → Processors and memory architectures**.

Keywords Memory Systems; Privacy; Oblivious RAM.

ACM Reference Format:
Chandrasekhar Nagarajan, Ali Shafiee, Rajeev Balasubramonian, and Mohit Tiwari. 2019. ρ : Relaxed Hierarchical ORAM. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3297858.3304045

1 Introduction

A growing amount of data storage and computation is migrating to the cloud. As a result, many different applications belonging to different entities are now executing together on an infrastructure that is controlled by yet another entity. This poses a number of concerns regarding security and privacy. Several attack models exist in such settings and various solutions have been designed to withstand these attacks [3, 5, 21, 23, 30, 32, 36, 37, 41]. For example, attacks can be launched by an untrusted cloud operator that has physical access to cloud servers. Such a cloud operator can swap commodity memory modules with custom memory modules that can snoop or modify data and addresses. A body of work [12, 42] even assumes that a client always sends data to the cloud in encrypted format, and builds protocols to enable cloud-based computations on encrypted data, i.e., the cloud never sees the user's data in unencrypted form. This highlights the potential paranoia associated with some cloud-based infrastructures.

Building a secure cloud infrastructure requires advances on several fronts, each targeting specific attack models. Here, we focus on attacks that are based on observable memory access patterns, and that have received attention from many prior works [10, 11, 26, 29, 34, 37, 45].

While a processor may offer high levels of security within the chip, exchanging data with entities outside the chip can pose vulnerabilities. As a baseline security measure, we assume that data will be encrypted before it is sent out of the processor. However, even though data is encrypted, memory addresses are typically not encrypted. This is because

traditional commodity memory systems are not equipped to perform decryption on memory devices. The plaintext addresses on the memory bus are therefore visible to an untrusted cloud operator. By examining the application's memory access patterns, an attacker with knowledge of the domain can decipher large amounts of information [14, 18, 45]; for example, it may be possible to reverse engineer a proprietary algorithm, or identify if a certain genomic population is more prone to certain diseases. It may even be possible to manipulate an application into revealing its secrets through its memory access pattern.

Indeed, interest in the above attack model dates back to the 1980s, when the notion of Oblivious RAM (ORAM) was formulated [10, 11]. In an Oblivious RAM model, an application's memory access pattern is buried within a larger memory access pattern such that an attacker cannot isolate the application's memory accesses, and any two access patterns are indistinguishable to an observer. The ORAM concept has been refined over many years [29, 34, 37, 45], primarily to reduce its bandwidth overheads. As a demonstration of the feasibility of ORAM, the Path-ORAM algorithm [37] was implemented with modest performance overheads on an FPGA system with high memory bandwidth and parallelism [26].

In spite of these advances, ORAM continues to impose orders of magnitude bandwidth penalties, which impacts performance in servers that are already memory-constrained. To reduce these overheads, we introduce a hierarchical ORAM composed of a relaxed smaller ORAM and a conventional full ORAM. The relaxed hierarchical ORAM (ρ) borrows principles from the full ORAM to guarantee privacy, but makes a number of design choices that lower capacity and reduce bandwidth overheads. We observe that these choices are not easily applicable to a regular full ORAM. We also discuss the effects of ρ as a potential timing channel leak and provide a solution to mitigate that concern.

We discuss the specifics of the ρ design in the following sections. To give a brief overview, our baseline full ORAM (Freecursive ORAM) has a capacity of 32 GB and requires $R \times (Z+1) \times L \times 2$ memory blocks per access, where $R = 1.55$ is the average number of recursive accesses, $Z = 4$ is the bucket size (the +1 with Z is for fetching metadata), and $L = 28$ is the depth of the ORAM tree. Meanwhile, ρ augments the LLC by offering relatively fast access to 2 MB of data (or more) per core, while requiring $Z \times L \times 2$ memory blocks per access, where parameters $Z = 2$ and $L = 17$ are determined empirically. Note that a ρ access eliminates any recursive look-ups and the overhead of fetching a separate metadata block per bucket (compact access).

While the paper focuses on a hardware implementation of ORAM in DDR memory, the general concept of a 2-level ORAM, where the first level can leverage a variety of efficiency techniques, can apply to other incarnations of ORAM as well. The ideas (e.g., compact metadata placement) and observations (e.g., trade-offs in tree depth and stash overflow) in this paper may therefore be broadly applicable.

2 ORAM Background

2.1 Attack Model

We examine potential attack scenarios where an untrusted cloud operator has access to the physical server. While signals within each package on the board are not observable, an adversary can observe signals being placed on various buses on the board. Each entity on the board can exchange data in encrypted form, but in the particular case of the memory system, a portion of the message (the address) has to be sent in unencrypted form. This is because commodity memory chips are heavily optimized for low cost. There is no decryption logic on commodity memory chips and they are essentially *passive* devices. On a write, they simply receive address and data on their pins and move the received data to the specified address (and vice versa for a read).

The side channel introduced by an exposed memory bus can be exploited in many ways. Prior work [18, 45] has demonstrated attacks based on memory address leakage. An adversary can snoop the access patterns on the memory bus to identify the control flow of the program [45]. These statistics can be potentially used to expose security related attributes like encryption keys (by following branching decisions) or to identify a propreitary algorithm (by matching the control flow graph against known algorithms). Similarly, the data access pattern can be compared to other known data access patterns to determine the nature of queries currently being processed [18]. For example, during genomic analysis, the distribution of accesses may reveal the chromosome being analyzed and hence the likelihood of certain ailments [38]. A third possible attack may employ variants of the Spectre attack [22]. In Spectre, a program was manipulated such that its internal secrets were converted into cache indices, which were then extracted with cache timing channels. Future attacks may extend the Spectre approach and instead manipulate the program such that internal secrets are converted into specific memory addresses, which are then observed on the memory bus.

In addition to the above vulnerability (an exposed memory bus), other hardware/software side channels may reveal an application's access pattern at various granularities. For example, in many systems, it is straightforward for a malicious OS to track the pages touched by an application. The side channel mitigation techniques being introduced here may apply broadly, but we will only focus on fine-grain leakage through cache block access patterns on a commodity DDRx memory bus.

The above vulnerabilities have fueled a large amount of research activity in the area of Oblivious RAM [10, 11, 37]. While our focus here is on ORAM approaches, we recognize that there may be other ways to guarantee address trace

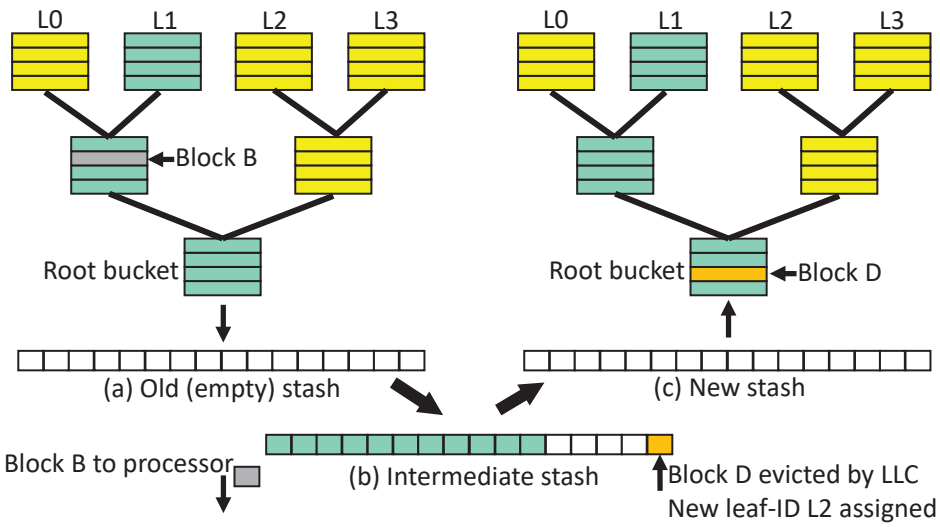

Figure 1. ORAM access protocol – Block B, associated with leaf-ID L1 is being accessed. (a) The path from root to L1 is fetched. (b) B is sent to the processor, while other valid blocks are kept in stash. Block D is evicted from LLC and placed in stash with new random leaf-ID L2. (c) The stash is drained and blocks (including block D) are written to the path from root to L1 while preserving the ORAM invariant.

confidentiality [1, 2]. Emerging active memory devices, e.g., Micron's Hybrid Memory Cube [20], may incorporate encryption/decryption units so that memory addresses can also be encrypted. Such approaches may open up other vulnerabilities (key management, leakage through power profiles, etc.), so this remains an open area of research. Further, such *active* memory devices are expected to be significantly more expensive [15] and may find limited use in large cloud/datacenter installations that manage large data sets – for example, a 2GB HMC device currently retails for over a thousand US dollars [15]. Current trends indicate that devices like the HMC will be used to implement a few-gigabyte DRAM cache [35], while the remaining hundreds of gigabytes of required memory capacity will be provided with low-cost commodity (passive) memory devices. The larger commodity memory system will require ORAM for privacy.

To summarize, we assume that the processor is the only trusted component and last level cache (LLC) misses are serviced on an untrusted DRAM memory system. We assume that an attacker is capable of eavesdropping on the memory bus and can also tamper with data stored in the untrusted memory. We incorporate the definitions of privacy and integrity of ORAM systems as defined in prior work [7]. Privacy is guaranteed if for any two sequences of requests from the CPU, the two resulting ORAM memory address sequences are computationally indistinguishable. Integrity is guaranteed if data returned by the memory system always matches the last block written to that location by the processor.

2.2 ORAM Basics

ORAM was conceived by Goldreich [10] and steady improvements have been made in the past few decades. At CCS 2013, Stefanov et al. [37] introduced the Path-ORAM algorithm and demonstrated an implementation of Path-ORAM on a Convey HC-2ex FPGA-based platform (PHANTOM [26]). While implementable, the design has a bandwidth requirement that is significantly higher than that of the native application without ORAM support.

ORAM Access Protocol

A Path-ORAM system organizes its memory blocks (cache lines) in the form of a full binary tree. Every node in this tree is a 'bucket' that can hold one or more blocks. The number of blocks per bucket and the number of levels in the tree are denoted by Z and L respectively. Each bucket also has a block that stores metadata for the data blocks in that bucket.

When the processor requests a block B, a *Position Map (PosMap)* table is accessed to determine the leaf-ID corresponding to that block. The example in Figure 1a shows that block B is associated with leaf-ID $L1$. Next, blocks in all the buckets from the root to leaf-ID $L1$ are fetched from memory. This brings block B, a number of 'dummy' blocks, and several other valid blocks from memory to processor. Block B is sent to the processor and LLC, while all other valid blocks are placed in a 'stash' on the processor (Figure 1b). This is followed by a stash drain, where many blocks from the stash are written to blocks in the path from root to leaf-ID $L1$, while preserving the invariant that every block lie on the path from root to its own leaf-ID (Figure 1c). When blocks fulfilling this condition are not present in the stash, dummy blocks are written instead.

When the LLC evicts a block, it is placed in the stash and assigned a random leaf-ID (in the example in Figure 1b, block D is assigned to leaf-ID $L2$). This allows a block accessed by the processor to later move to a new location in memory. This shuffle is vital in realizing an indistinguishable memory access pattern. When the processor makes a request for block B, the attacker sees blocks being read from many locations and then re-encrypted blocks being written back to the same locations. When B is accessed again, blocks are fetched from a different set of locations (since B is now associated with a different leaf vertex). From the attacker's perspective, it is impossible to detect that B has been touched again.

Design Details

Path-ORAM organizes the memory into data buckets, each with Z data blocks. While early work assumed large block sizes in the kilo-byte range [26], design space explorations [7, 29] have shown that performance is optimized with 64-byte blocks and $Z = 4$, i.e., each bucket has 4 64-byte data blocks and 1 metadata block. This helps balance the probability of stash overflow and the number of blocks fetched on every ORAM access. When the stash does overflow, a few dummy reads/writes are performed to drain the stash [29].

The PosMap has an entry for every block in memory, i.e., it is a large structure that may not fit in the processor's LLC. Fletcher et al. [7] designed a recursive ORAM where the PosMap is itself placed in an off-chip ORAM which is referenced by a cached table called a PosMap Lookaside Buffer (PLB). This recursion based Path-ORAM design is the state-of-the-art baseline assumed in this study. We assume a 64 GB memory system (effectively 32 GB capacity because half the blocks are dummies), implemented with block size of 64B, bucket size Z of 4, and 28 levels in the binary Path-ORAM tree. We refer readers to the Path-ORAM and Freecursive ORAM papers for more details [7, 37].

3 Proposal

The previous section describes our baseline Full ORAM structure that adopts most of the innovations from recent work [7, 26, 29, 37]. This Full ORAM uses recursion to store large PosMaps in ORAM, while avoiding some of these recursive look-ups with a PosMap Lookaside Buffer (PLB). As is common in large-scale installations, we assume an ECC memory [16], i.e., every 64-byte data block is accompanied by 8 bytes that are used for error detection and correction.

3.1 ρ Overview

We are effectively implementing a large cache before the Full ORAM, thus reducing the overheads imposed by a Full ORAM. This large cache augments the on-chip LLC (an exclusive hierarchy), but has to be placed in memory because it is too large to fit on the processor. To avoid information leakage through the memory system, this cache must be implemented as an ORAM. We will refer to this in-memory cache as ρ, the Relaxed Hierarchical ORAM. Given its small size, relative to the Full ORAM, it can be designed for bandwidth-efficiency, thus making it significantly cheaper to access than the Full ORAM. Figure 2 provides a high-level overview of the proposed design.

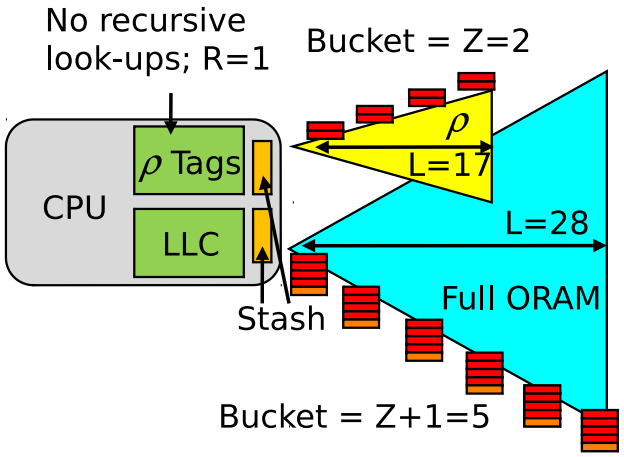

Figure 2. Memory hierarchy overview of ρ.

When accessing ρ, an on-chip set-associative tag look-up is first performed to detect a hit. On a hit, given the leaf-ID associated with the tag, the entire path from root to leaf-ID is fetched from ρ's off-chip data array.

We first allocate a portion of the processor chip's real estate to implement tags for ρ. A ρ tag look-up indicates if the processor should look up ρ or the Full ORAM. As shown in Figure 3, a portion of the processor's LLC tag and data arrays are allocated for ρ tag storage. For example, the baseline processor's LLC is adjusted from 2 MB to 1.75 MB (sizes representative of what may be required per core). This allocates 256 KB for ρ on-chip tags, that can support a ρ capacity of 2.5 MB in off-chip memory, assuming that each tag is 50+ bits wide and supports a 64-byte block (more details in Section 3.2).

The data blocks of the 2.5 MB ρ are implemented as an ORAM with $Z = 2$ and $L = 17$. Since a small Z increases the probability of stash overflow, we balance it out by assuming low block utilization U [29], i.e., for every data block, there will be more dummy blocks. Including the metadata block per bucket, the total space occupied by ρ may be 24 MB (a small fraction of off-chip physical memory), of which only 2.5 MB is useful and has corresponding entries in the ρ tags.

As mentioned in Section 2, there are 3 major causes of bandwidth overhead in ORAM: blocks in a bucket $Z + 1$, depth of the tree L, and number of recursive look-ups R. Thus, for ρ, as shown in Figure 2, we have reduced the values of all four elements that lead to high ORAM overheads.

L has been reduced because of the smaller capacity, and Z has been reduced based on empirical tests for stash overflow. R has been reduced from a number greater than 1 (function of PLB miss rate) to exactly 1 (since we have on-chip tags for ρ). The fourth remaining overhead is the "+1", the metadata block that must be fetched from every level of the tree. As described subsequently, we introduce a *ρ-compact* layout, that leverages the ECC fields in every block to capture metadata, thus targeting this fourth overhead as well.

3.2 Design Details - ρ implementation

ρ Tag Array

We assume that the LLC and ρ form an exclusive hierarchy. Figure 3 shows the partition of the LLC into a smaller LLC and a tag array for ρ. In modern CPUs, each LLC data block is 64B. This 64B space, including the corresponding 32b tag array counterpart, is re-purposed as tag entries for one set in ρ. Each tag entry has the following fields: (a) 32 bits for address tag, (b) 1 bit for NRU replacement policy, and (c) n bits for leaf ID in ρ.

The data array of ρ is organized as an ORAM tree, including many dummy blocks. Each tag entry points to a valid block stored in ρ. During a ρ look-up, the tags in a set are accessed. If no match is found, a Full ORAM access is performed. If a match is found, we use the leaf ID to fetch an entire path in the ρ ORAM tree. The tag array essentially serves as the PosMap structure for ρ.

A single ρ set can contain $(512+32)/(33+n)$ ways. For $n \leq 20$, a 10-way ρ set can be implemented. As another example configuration, consider an LLC with capacity of 2 MB, of which, 1 MB is allocated for ρ tags. The ρ tags (10 ways, 16K sets) can point to data blocks that have a total capacity of 10 MB. To reduce stash overflow rate in ρ, we assume low block utilization U, i.e., the 10 MB of data is accompanied by a large number of dummy blocks. The total capacity of the ρ data array may therefore be 64 MB; with $Z = 2$, this corresponds to 256K leaf buckets and a value of $n = 18$. Note that the 64 MB occupied by ρ is only 0.1% of a 64 GB physical memory, i.e., it has a negligible impact on physical memory capacity.

ρ Data Array

The data array of ρ maintains the following fields per block: (a) 64B for the data block, (b) $L-1$ bits for leaf ID, (c) 4B for address tag, (d) 4B plaintext counter for encrypting other fields, (e) 1 bit valid flag, and (f) 56-bit VMAC for integrity verification [13]. Assuming that L is in the neighborhood of 20, and $Z = 2$, each 64B data block requires nearly 18 bytes of metadata. Since commodity memory systems must fetch blocks at 64B block granularity, each bucket contains Z 64B data blocks, plus one 64B metadata block. The 18-byte overhead can be reduced, e.g., by having one VMAC per bucket, but because of the granularity constraint, it doesn't reduce the amount of data that must be fetched.

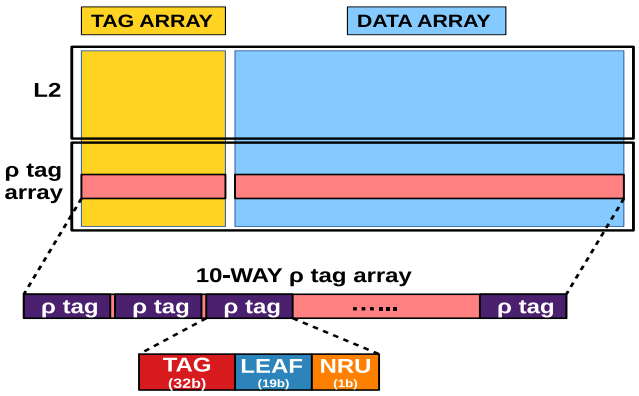

Figure 3. Partitioning of Last Level Cache.

Accessing ρ

ρ and the Full ORAM are exclusive with respect to each other. We assume that there is enough space to hold 200 elements in ρ's stash. Whenever a block is not found in ρ's stash or in ρ, it is fetched from the Full ORAM. The fetched block is removed from the Full ORAM's stash and placed in the LLC. The block evicted out of the LLC is written into ρ's stash. In parallel, ρ performs a full path access of its own to fetch a victim block for eviction. The victim block is placed in the ORAM stash. In other words, the use of ρ does not impact stash overflows in the Full ORAM. If the requested block is a ρ hit, the leaf ID found in the ρ tag array is used to initiate a full path access in ρ from leaf to root. The requested block is found among these fetched blocks by checking the accompanying metadata and tags. The requested block is sent to the LLC and the block evicted by the LLC is placed in ρ's stash.

3.3 Design Details - ρ-Compact Access

While the lower Z, L, and R are conceptually easy to understand, the reduction in metadata fetches is more complex. The extra metadata block fetched from every bucket is a significant overhead when Z is small. To avoid fetching a separate metadata block, we propose a memory layout that places the ORAM metadata in the space used for ECC. We must first solve two challenges. First, we must preserve the same level of error tolerance. Second, we must reduce the size of the metadata field. For the rest of this discussion, we assume a bucket size of $Z = 2$.

3.3.1 A Distributed ECC Layout.

To address the first challenge, we use a two-level ECC technique that separates the error detection code (EDC) from the error correction code (ECC). Error detection is performed with a 24-bit CRC code that is placed in the 64-bit ECC field associated with each block in the baseline. This EDC code is strong enough to detect errors with a very high probability. When an error is detected, a separate error correction step

must be performed. Error correction is performed with the standard 64b Hamming SECDED (Single Error Correct, Double Error Detect) code per block. These codes are placed elsewhere though. We place the SECDED code for all blocks in a ρ path alongside the leaf bucket (see Figure 4). For example, in a 17-level ρ, we would have to store $17 \times 64b \times Z$, i.e., 272 bytes. These codes therefore occupy 5 cache blocks. These 5 blocks need not be fetched when reading the path. They are accessed only if the EDC indicates an error. However, when the path is written back, the 5 ECC blocks would have to be updated. In basic ρ, a ρ access (with $Z = 2, L = 17$) involves 51 block reads and 51 block writes; with the compact layout, a ρ access involves 34 block reads and 39 block writes.

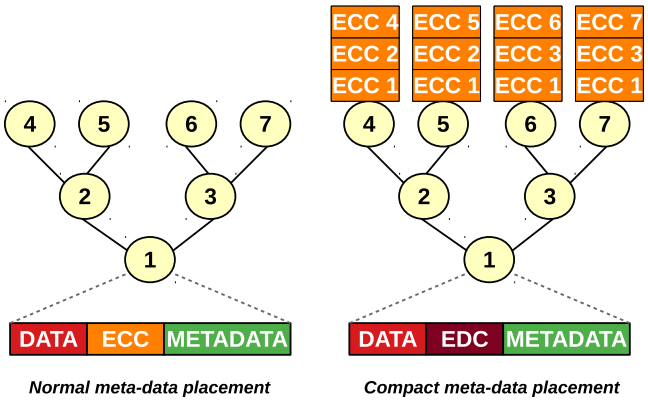

Figure 4. Compact layout to reduce metadata fetch overhead.

As seen in Figure 4, the ECC codes for a block are associated with all descendant leaf nodes. For example, the root node 1 has its ECC code (ECC 1 in Figure 4) attached to all leaf nodes. Each ECC code is updated when that leaf is updated, i.e., only the most recently updated ECC code for a block is correct and the other copies are stale codes for earlier versions of that block. For error recovery, we must find the most recent code for a block. To facilitate this, every written path includes a 64-bit version number that is included in its 5 ECC blocks, increasing the storage requirement from 272B to 280B. The CPU maintains this 64-bit global counter and increments it on every path write. This version number is large enough that it will not overflow during the system's lifetime. When a block error is detected, the version numbers of all descendant leaves for that block are read. The SECDED code associated with the largest version number is then used for error recovery. While this error recovery is expensive, it is invoked rarely, given today's DRAM error rates [24]. This distributed ECC layout does not introduce new information leakage – the access patterns remain deterministic and indistinguishable. Error recovery steps betray the location of an error, not the properties of an application.

3.3.2 Metadata Compaction.

Next, we address the second challenge. In the baseline, each block has an attached 64-bit SECDED code. Thus, in one bucket with $Z = 2$, 128 bits that were used for the SECDED code can be now used for ORAM metadata. We have consumed 24 of those bits for the EDC (see Figure 5). We now need to squeeze the ORAM metadata into the remaining 104 bits. Recall from the earlier discussion in Section 3.2 that every baseline block has nearly 18 bytes of metadata associated with it, so we use the following techniques to engineer a more compact metadata organization.

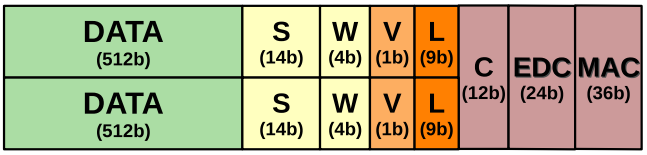

Figure 5. Organization of two blocks in a bucket ($Z = 2$).

1. We replace the 4-byte tag per block with two fields: set number and way number in the ρ tag array (represented by S and W in Figure 5). These occupy 14b and 4b respectively for the example ρ we've been discussing. This is enough to identify the block that we're looking for. There is also a valid bit per block to distinguish a valid block from a dummy block.
2. We reduce the leaf ID bits from $L-1$ to 9 bits. When a block resides in level K of the ORAM tree, the K most significant bits of its leaf are known. Therefore, for the last 9 levels (close to the leaves), 9 bits are enough to represent the rest of the leaf ID. For levels close to the root, we store the additional required leaf ID bits in an on-chip table. Even for a larger 19-level ρ, this translates to $10 \times 2^{10}b$, which is a small 1.25 KB table.
3. Instead of a 4B counter per bucket, we now use a 12-bit local counter. This 12-bit local counter is concatenated with a larger global counter on the processor to produce the final counter that is used during encryption. Every time a local counter overflows, we must increment the global counter, and conseqeuntly re-encrypt the entire contents of ρ. Such a re-encryption would be very expensive for a Full ORAM, but is a much smaller overhead for ρ given its much smaller size. Since buckets near the root see more accesses and more local counter increments, they tend to overflow more often. To alleviate this effect, the top 10 levels of the tree are provided 10 extra bits for each local counter. Similar to the leaf-ID bits, these additional local counter bits are stored in a 1.25 KB table on the processor. The number of accesses between two overflow events is typically well over a million (given 22-bit local counters for top levels). That is, in the worst

case, after every million ρ accesses (73 million block accesses), we have to read and write the entire contents of ρ (288K block accesses). This is a negligible overhead in terms of overall execution time, but it does introduce occasional latency hiccups.

4. With the above techniques in place, we only have room for 36 bits of VMAC per bucket (the baseline has 56 bits, similar to SGX). The remaining 20 bits must therefore be included in the ρ tag array. We must therefore reduce the associativity of ρ from 10 to 8.
5. The VMAC, EDC, and counter can be shared by both blocks in the bucket since they are always accessed together. Figure 5 shows that these 3 fields are shared, while the other fields are private to each data block.

We have thus engineered a solution that packs ORAM metadata into the space typically occupied by SECDED codes. This reduces the bandwidth requirement per bucket from 3 to 2 blocks, a significant reduction.

3.4 Why do Similar Ideas Not Apply to the Full ORAM?

ρ has been designed so that each of the four terms in the ORAM bandwidth equation have been reduced. It is natural to wonder why similar reductions cannot be directly applied to the full ORAM. Clearly, the full ORAM inherently has higher depth L than ρ. The full ORAM also has a large PosMap that does not fit on chip; so it has a recursion factor $R > 1$. While the full ORAM could have used bucket size $Z = 2$, it would have to use low utilization U to keep stash overflow under check. This large drop in effective memory capacity is likely not palatable. Finally, the optimizations used in Section 3.3 to create a compact layout do not scale and would lead to very high overheads if applied to a full ORAM, e.g., we were able to compact the metadata because some of it could be moved into 1.25 KB on-chip tables.

3.5 Security Analysis

In the baseline ORAM, the misses emerging from the LLC exhibit the indistinguishability property. With the proposed hierarchy of ρ and a Full ORAM, the misses emerging from the LLC and the misses emerging from ρ, both separately exhibit the indistinguishability property since they both separately employ the Path ORAM algorithm. However, it is well known that the baseline ORAM is vulnerable to a couple of information leaks; those leaks are amplified by a deeper hierarchy, and we must show that known mitigation techniques are equally effective for the ρ hierarchy.

Timing Channels.

In a baseline ORAM implementation, there is a potential threat of leaking information about application memory intensity and cache miss rates through the ORAM access rate. The deeper hierarchy introduced here exacerbates that information leakage by exposing cache miss rates for both the last level cache and for ρ. For applications that are sensitive to such timing channel leaks, we demonstrate that such leaks can be easily eliminated. The ORAM controller can issue alternating requests to either ρ or the Full ORAM based on a per-application frequency setting that is input-independent. If one of the ORAMs does not have a pending operation, a dummy access is issued. We discuss the effects of this policy in Section 5.7.

In the timing-channel sensitive implementation, we empirically determine (and preset) the schedule of accesses between the Full ORAM and ρ . If the access rate for the Full ORAM and ρ is set to m and n respectively, the trace observed on the memory bus would appear as a repeating pattern of $[n * access(\rho), m * access(FullORAM)]$. Note that $access()$ denotes the series of memory accesses performed for either ρ or the Full ORAM.

This leads to a deterministic sequence of accesses by the memory controller, with zero information leakage about the input-dependent miss rate at either the LLC or at ρ. A similar deterministic schedule of ORAM accesses has also been used in other ORAM protocols to protect against timing channels [1, 2, 8]. We observe that a similar approach (statically defining the access rate for ORAM) can be effectively extrapolated to a 2-level hierarchy. It is possible that a poor choice of n and m may yield worse performance than the baseline ORAM – this is analyzed later in Section 5.7.

Stash Overflow.

A second potential problem in the baseline ORAM protocol is stash overflow because it can leak information and cause inefficiencies. We must answer if this problem is potentially amplified in the proposed architecture because both ρ and the Full ORAM have separate stashes.

We will first address the efficiency issue. Prior work [7–9, 29] has shown that a background stash overflow mechanism can dramatically lower the stash overflow rate. We adopt a similar mechanism here for both ρ and the Full ORAM, offering the same worst-case stash overflow rate guarantees as prior work for each level of our hierarchy. When an overflow is imminent, as determined by a stash high/low water mark, we start writing back blocks from the stash into the ORAM. Instead of a randomized leaf selection, we use a deterministic path selection to flush out stash elements [9], where leafIDs are selected in a reverse lexicographic order. While stash overflows tend to be more frequent in ρ because of its lower Z, stash overflows in ρ are also alleviated by the periodic re-encryption process when a ρ local counter overflows. During this re-encryption, all blocks are placed in leaf nodes. Different applications exhibit different stash overflow rates and overheads. We empirically observed that for all our benchmark programs, such stash overflow management has a negligible performance impact in ρ.

Stash overflows can also leak information since different applications and access patterns exhibit different stash overflow rates. This leakage can be easily eliminated because

the stash overflow mechanism appears exactly like a regular ORAM access and further, we employ a deterministic schedule of accesses to ρ and the Full ORAM (introduced to eliminate timing channels). Thus, high or low stash overflow rates are indistinguishable.

Summary.

The observable memory access pattern is: $[n * access(\rho), m * access(FullORAM)]$. The access to each ORAM follows a deterministic sequence, revealing nothing about the application's cache locality or its stash overflow rate. Each individual ORAM access is a path to a random leaf node in that ORAM, thus preserving the indistinguishability guarantee of the baseline ORAM.

4 Methodology

We use trace-based simulation for our evaluation. Our traces are obtained by running benchmarks from SPEC2006 using the Simics [6] full system simulator. These traces are generated after fast-forwarding to the region of interest and warming up the caches. These traces are then fed into USIMM [4] for cycle-accurate simulations of the DRAM memory. The DRAM device model and timing parameters have been obtained from Micron datasheets and are summarized in Table 1 with other Simics simulation parameters. We adopt the open-page address mapping policy that places consecutive cache lines in the same row [19]. The memory controller scheduler employs the FR-FCFS policy [31]. It uses high/low water marks in the write queue to drain writes in batches [4].

Processor	
ISA	UltraSPARC III ISA
CMP size and Core Freq.	1-core, 1.6 GHz
Re-Order-Buffer	128 entry
Fetch, Dispatch, Execute, and Retire	Maximum 1 per cycle
Cache Hierarchy	
L1 I-cache	32KB/2-way, private, 1-cycle
L1 D-cache	32KB/2-way, private, 1-cycle
DRAM Parameters	
DDR3	MT41J256M4 DDR3-1600 [17],
Baseline DRAM Configuration	2 72-bit Channels 1 DIMM/Channel (unbuffered, ECC) 4 Ranks/DIMM,9 devices/Rank
DRAM Bus Frequency	800MHz
DRAM Read Queue	48 entries per channel
DRAM Write Queue Size High/Low Watermarks	48 entries per channel 32/16

Table 1. Simulator parameters

We model the L2 cache (LLC) in USIMM with 2 MB for a single core as a baseline for all simulations. The L2 sizes are varied to accommodate the ρ tag structure. We experiment with different sizes for the LLC and the ρ tags to find the optimal allocation. The LLC is warmed up with 3 million L1 misses before starting performance measurements. All simulation results are obtained for one million L1 miss requests, which is long enough to measure many millions of DRAM accesses to the two ORAMs.

Full ORAM parameters	
Data Block Size	64 B
Levels	28
PLB Size	64 KB
Cache lines per bucket (Z)	4
ρ parameters	
Data Block Size	64 B
Levels	17-19

Table 2. ORAM Configuration

Our baseline Full ORAM is a Freecursive model with 28 levels and 4 64-byte data blocks per bucket. For ρ, we assume 64B blocks, and explore the design space for L, Z, U.

We modified the USIMM DRAM simulator to implement Freecursive ORAM and ρ. To decrease the latency per ORAM access, the binary tree is laid out as multiple k-ary subtrees which appear as contiguous locations in DRAM rows. These subtree based layouts ensure that the number of rows opened per ORAM access is reduced. The bucket size decides the arity and the subtree heights for this configuration. We adopt a mapping scheme as suggested in [29] with bit addressing in the sequence of - $channel : column : rank : bank : row$.

Our stash eviction process is similar to that of Path ORAM. When blocks are brought in, the block specifically requested by the processor is placed in the upper level cache; when evicted by the upper level cache, it is placed in the stash along with a randomly selected new leafID. Other blocks not specifically requested by the processor are placed directly in stash with no update to their leafID. Note that all blocks are re-encrypted before being written back.

5 Results

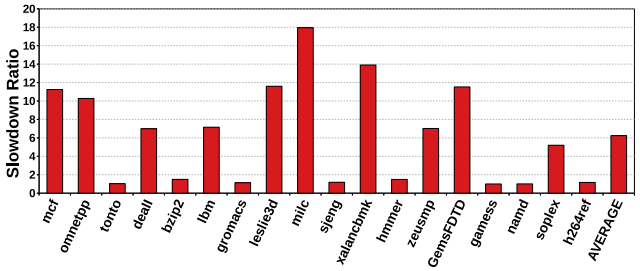

Figure 6. Slowdown of Freecursive ORAM against non-secure baseline.

5.1 Comparison with Non-Secure Baseline

We simulated a Freecursive ORAM design with a 2 MB LLC, while caching the first 10 levels of the ORAM tree. Against a non-secure baseline, the slowdown due to the Freecursive

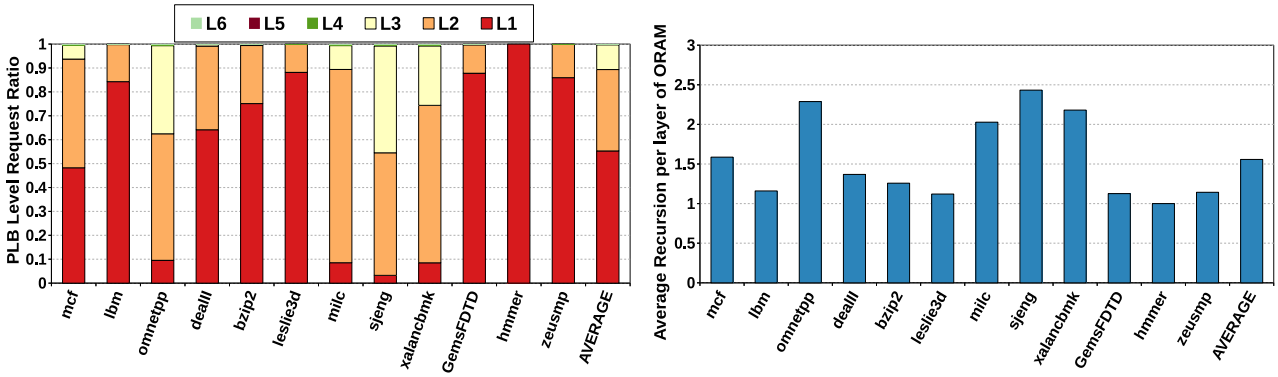

Figure 7. (a) Ratio of requests to every level of PLB (b) Recursion factor for every benchmark

ORAM implementation is reported in Figure 6. This slowdown is caused by the many memory accesses required by the ORAM protocol instead of a non-secure single memory access. While most of the benchmarks suffer from a significant performance loss, some show only a slight reduction (e.g., *hmmer*) due to a high L2 hit rate. Freecursive ORAM with 2 channels shows an average slowdown of 6.2×.

Figure 7 shows the hit rate in different levels of PLB in Freecursive ORAM. Some applications require more than one access to the PLB (e.g., *mcf*, *omnetpp*, *milc*, and *sjeng*), while others do not need recursive lookups for most of their requests. We observe that the first three levels of the PLB are accessed the most. As shown in Figure 7b, each request requires an average of 1.55 recursive ORAM accesses to fetch the requested block.

5.2 Optimal Cache Partition Selection

We have identified three parameters that define the best design point: on-chip capacity, optimal bucket size, and in-memory capacity. The on-chip capacity captures the performance of the last level cache for different applications. The bucket size affects the overflow handling overhead of ρ. The last parameter is dependent on the levels L and bucket size configuration of ρ. If the in-memory capacity of ρ is $C = (2^L) \times Z$, and the number of valid blocks in ρ is X, we can define utilization to be $U = X/C$. Once the on-chip cache capacity is determined, X is fixed; the utilization (U) is then controlled by the levels L in the ρ tree. We will progressively freeze each of these parameters in our analysis.

The first step is to find the optimal cache partitioning for LLC and ρ. The next set of experiments are performed using the default ρ configuration, i.e., a 10-way ρ organization and without the compact layout. In this experiment, we fix the bucket size to $Z = 4$ and the tree depth to 18 levels. This is an effort to finalize an optimal L2 cache partition with a good hit rate and without any chances of overflow (guaranteed with $Z = 4$). We then varied ρ's tag array partition sizes to 256KB, 512KB, 1MB, and 1.5MB while allocating the remaining (of 2MB) for the LLC.

Figure 8 shows the speedup over Freecursive ORAM for different partitions. The improvement varies, based on benchmark working set size. Benchmarks like *xalancbmk* show around 50% improvement with these configurations. However, benchmarks like *bzip2* and *h264ref* suffer for all of our partition choices. The configuration that allocates 256 KB for ρ tags yields the best average performance and is often the best design point for each benchmark. For the rest of our analysis, we fix the LLC partition at 1.75 MB and the ρ tag partition at 256 KB. This fixes the value of X, the number of valid blocks in ρ.

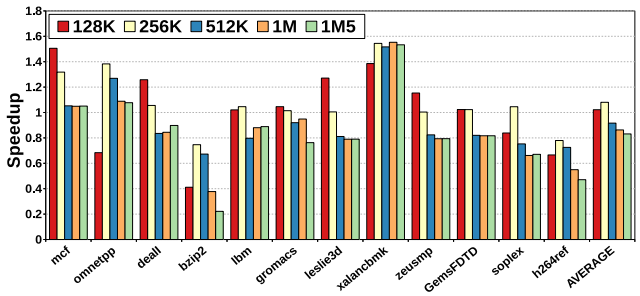

Figure 8. Speedup for different tag array sizes for ρ (Z=4, L=18), relative to Freecursive ORAM. The LLC is 2MB.

5.3 Impact of Depth/Utilization

Having fixed X, we are next going to evaluate the impact of the number of levels L on the performance of ρ. Note that while L increases the memory overhead of ρ, this is still a negligible fraction of the overall off-chip memory capacity. As the number of levels is increased, the bandwidth overhead per ρ fetch increases, but the rate of stash overflow decreases (since utilization U reduces). Figure 9 shows this analysis for $Z = 1$, i.e., the highest sensitivity to stash overflow. The baseline in this analysis is the best ρ organization ($Z = 4$ and 256 KB for ρ tags) shown in Figure 8. We observe that $L = 19$ yields the best balance between overflow and bandwidth overheads.

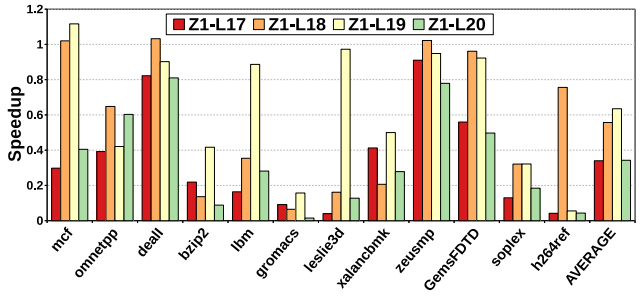

Figure 9. Impact of number of levels on ρ (Z=1).

5.4 Impact of Bucket Size

Next, we evaluate the impact of the bucket size on ρ's performance. For these experiments, we retain ρ's on-chip tag space of 256 KB, and keep utilization U constant as well; the values of L and Z are changed in tandem to keep U constant. The analysis is shown in Figure 10, normalized to the Freecursive baseline. We observe that $Z = 1$ is highly sub-optimal because of frequent stash overflows and that $Z = 2$ is able to bring these overflows down to a tolerable rate. Additional increases to Z have a minor impact on overflows, but reduce performance by increasing the bandwidth demands for every ρ access. Thus, Figures 8 and 10 show two different views of the overflow/bandwidth trade-off in our design space exploration; the former graph also factors in the impact of utilization U. The low value of the optimal Z is the motivation for the compact layout; it helps bring down the fetch overhead of a bucket from 3 to 2 blocks.

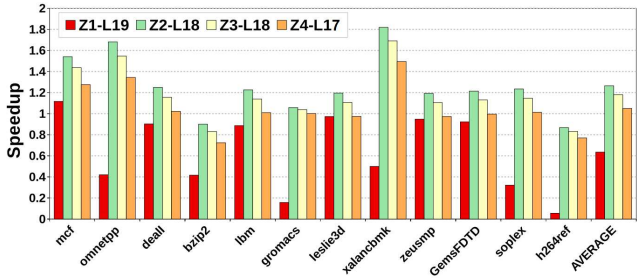

Figure 10. Effect of Z for a 256 KB ρ, normalized against Freecursive baseline.

5.5 Impact of Utilization

As an added sensitivity analysis, Figure 11 repeats the analysis of Figure 9, but with $Z = 2$. This figure is again exploring the trade-off between number of blocks being fetched per ρ access and the impact of utilization on stash overflow rate. Similar to the earlier analysis, the optimal design point has a depth $L = 19$. The fluctuations in this graph are more minor because $Z = 2$ has already brought the overflow rate to tolerable levels. After this design space exploration, we identify the optimal design point with on-chip tag space of 256 KB, $Z = 2$, and $L = 19$.

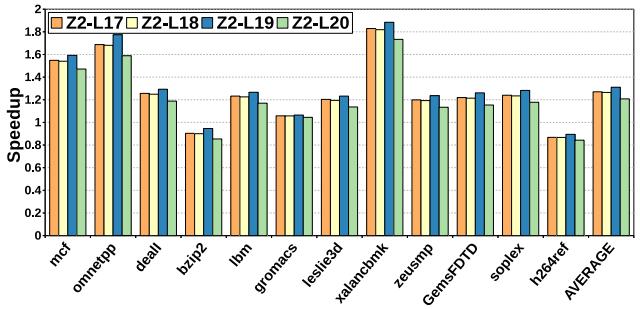

Figure 11. Varying utilization (in-memory capacity) for $Z = 2$, speedup relative to Freecursive.

5.6 Improvement from the Compact ρ Layout

We next evaluate the impact of the compact layout. Figure 12 shows a comparison of the best non-compact design point ($Z = 2$, $L = 19$) and compact layouts (indicated by ρC), as tree depth L is varied. We observe that highest performance is observed when tree depth is 17; this design point is 50% better than Freecursive ORAM and 20% better than the best non-compact ρ layout.

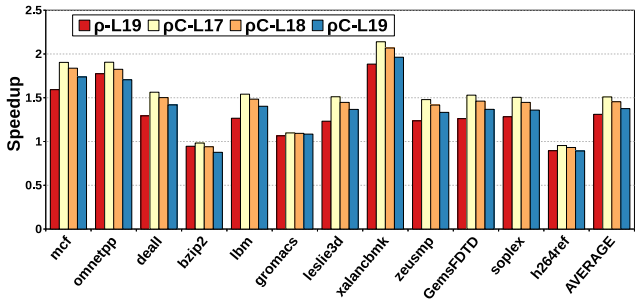

Figure 12. Speedup due to the compact ρ layout, relative to Freecursive.

Figure 13 shows the memory accesses in ρ for both the non-compact and compact layouts, normalized to the baseline Freecursive ORAM with a 2MB LLC. Benchmark $h264ref$ has a memory access ratio higher than one because it is penalized by the smaller 1.75 MB LLC in ρ. The average reduction in memory accesses is 19% for the non-compact layout and 30% for the compact ρ layout. The reduction in memory traffic and the higher performance should also result in lower power and energy profiles for ρ (not evaluated here).

5.7 Security from Timing Channels

As discussed earlier, the access rate for the Full ORAM and for ρ can reveal the locality behavior of the application. This

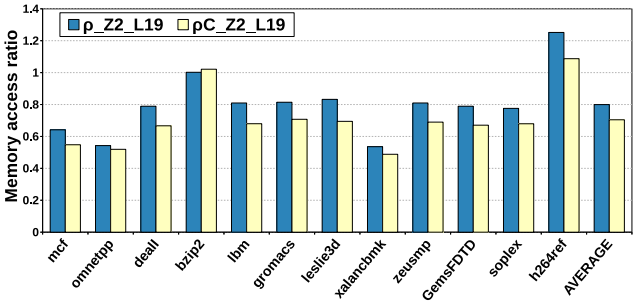

Figure 13. Memory accesses for non-compact and compact layouts, relative to the Freecursive baseline.

information leakage can be eliminated by constantly keeping the memory bus busy, i.e., every time a request is serviced, a new request is immediately processed. This new request may be a dummy if there is no pending request in the memory controller. Since the Full ORAM and ρ accesses have varying tree depths and can be distinguished, the memory controller must deterministically process n ρ requests before processing one Full ORAM request. Figure 14 shows speedup, relative to the Freecursive baseline, as n is varied from 1 to 4 for the ρ with compact layout. The optimal design point varies across the workloads based on their hit rates in ρ. On average, we see that $n = 2$ yields the highest speedup of 1.25. A poor choice of n, or the use of ρ can occasionally cause slowdowns, e.g., in bzip2, ρ does worse than the baseline for all non-zero values of n. This deterministic schedule is maintained even when dealing with stash overflows. Stash overflow handling is therefore indistinguishable from a regular ORAM access, thus eliminating another source of information leakage. An additional firmware layer that implements an epoch (and dynamic rate learning) based issue control [8] can be deployed on top of this technique to further improve throughput, while not allowing any input-dependent leakage.

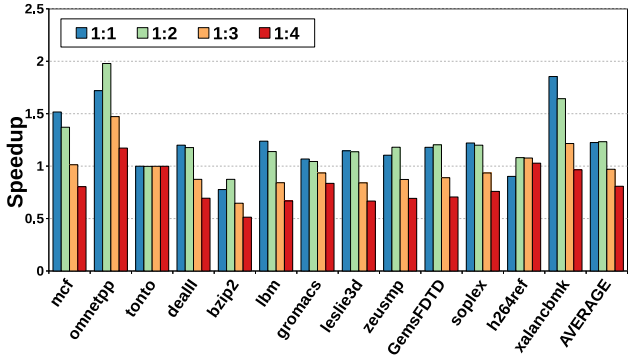

Figure 14. Varying access ratio for Full ORAM and ρ; speedup relative to Freecursive.

5.8 Sensitivity Analysis

Finally, as a sensitivity analysis, we evaluate our best ρ design point while varying the available memory bandwidth and LLC size. Figure 15 shows the impact of the best design point with a compact layout for 1 and 4 channel memory system configurations, relative to the Freecursive baselines with the same channel configurations. The trends are similar, primarily because ORAM systems almost always operate at saturated bandwidths. Figures 16 and 17 show the speedups for ρ with smaller (1MB) and larger (4MB) LLCs. We show results for three configurations, P1, P2, and P3, where 1/3, 1/2, and 2/3 of LLC capacity is allocated for ρ tags. With the small LLC, ρ benefits from a larger LLC allocation, while for the large LLC, ρ benefits from a smaller LLC allocation. Not surprisingly, the benefits with ρ are inversely proportional to the size of the total LLC.

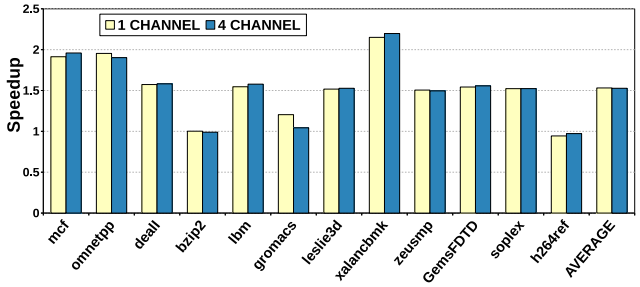

Figure 15. Effect of number of channels on ρ speedups.

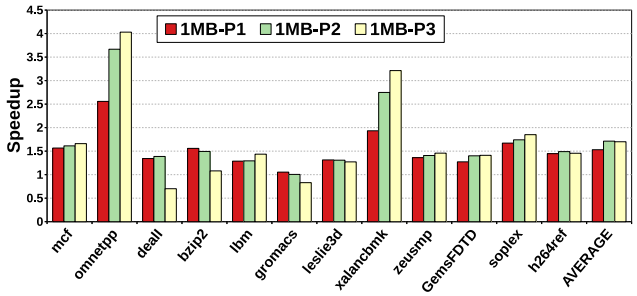

Figure 16. Performance of ρ with a 1 MB LLC.

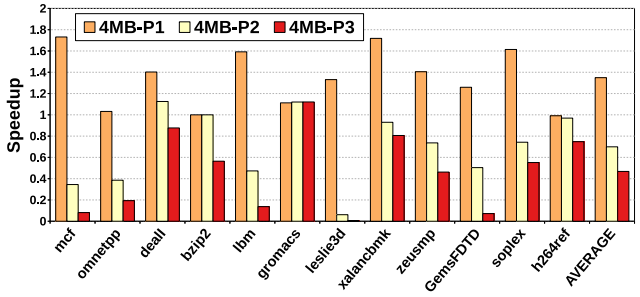

Figure 17. Performance of ρ with a 4 MB LLC.

6 Related Work

The concept of ORAM was first introduced by Goldreich and Ostrovsky for address obfuscation [11]. Since then, numerous works have proposed a variety of ORAM models to reduce the bandwidth and capacity overhead, and exploit memory parallelism [10, 11, 26, 29, 34, 37, 45].

There are many other proposals to improve ORAM performance. Ren et al. [29] explore the design space of ORAM, and propose background eviction and optimized data layout in memory. Yu et al. [43] take advantage of spatial locality in data and prefetching. Freecursive ORAM uses a PLB to avoid most of the recursive accesses to ORAM [7]. Fletcher et al. [8] eliminate side channels in ORAM-based systems and trade off performance for security. ForkPath [44] merges ORAM paths to avoid fetching redundant data blocks. They also introduce an on-chip caching of the ORAM subtrees that are repeatedly accessed in consecutive requests.

The idea of PrORAM [43] is to maximize locality in every ORAM tree access, essentially emulating a prefetching scheme and arguing that locality and obliviousness can coexist. While we do not currently take advantage of locality, future work can explore if the on-chip cache can be used to house metadata for a relaxed ORAM super-block scheme.

In Ring ORAM [28], Ren et al. reduce the bandwidth overhead of ORAM access by fetching a single block instead of the entire path from the tree. They employ a frequency based decision system to evict the elements in the stash which were populated using this type of access. Their protocol scans the tree for metadata to find the valid position of the block of interest. There is a potential to incorporate this design choice in ρ as well. The on-chip ρ tags could potentially hold an index pointer for every block's valid location in the tree's buckets. This can eliminate the need for a redundant path read for identifying the valid block of interest.

Ghostrider [25] is a compiler assisted hardware implementation for ORAM that assists in preserving Memory Trace Obliviousness. Since software managed codes can control the size of data in the secure region, they can leverage ρ to optimize their bandwidth.

Active memory components, such as HMC [20, 27], can provide address obfuscation if they have encryption/decryption logic in their logic layer [1, 2, 40]. However, leveraging active memory components requires trusting the memory vendor. This approach also offers limited memory capacity at a high cost that is incurred by every application that executes on the server, regardless of their security/privacy requirements. The work of Shafiee et al. [33] avoids active memory components, but uses logic chips on DIMMs to push parts of the ORAM protocol closer to memory and shelter the processor from a large fraction of the data movement required by ORAM.

Our work here is the first to propose a hierarchical ORAM structure, unlike the recursive PosMap structures seen in prior work. A key novelty in our work is the use of on-chip cache space to implement tags for an in-memory small ORAM. Another key innovation is the compact layout that reduces the overheads of metadata management. Similar to Ren et al. [29], we too perform a design space exploration to understand the impact of bandwidth, stash overflows, utilization, etc. on overall ORAM performance. We also observe that a hierarchical ORAM may be compatible with other secure systems like SGX [13, 39], where hierarchy is used to manage the overheads of integrity verification.

7 Conclusions

Attacks like Spectre have shown that secrets can be converted into cache footprints, and possibly other system footprints that can be exploited through side channels. One such side channel is an exposed memory bus. While an ORAM can close such a side channel, it imposes a significant performance penalty of over 6×. Therefore, it is important to advance the state-of-the-art in ORAM. This paper focuses on improving a hardware ORAM baseline by introducing another level to the hierarchy. We show that with the use of hierarchy, the first-level ORAM can introduce a number of relaxations and optimizations that would not be applicable to a second-level full ORAM. The relaxed first-level ORAM offers the same indistinguishability properties as the Full ORAM, but has data structures and metadata that offer lower bandwidth overheads and can tolerate occasional overflows in the stash and counters. For applications that exhibit a certain degree of locality, it is better to employ the proposed 2-level hierarchy than implement a single-level deep ORAM. In our evaluated benchmark suite, this was true for all but one application (bzip2). Our optimal design point offers performance that is 50% higher on average than that of the Freecursive ORAM baseline. With a deterministic scheduler that eliminates timing channels, the performance improvement is 25%. With hierarchical ORAMs and a deterministic scheduling policy, we not only preserve the indistinguishability property of the baseline ORAM, we also eliminate other leakage sources (cache hit rates and stash overflow rates). We believe that ρ represents a framework where additional future optimizations and approximations can be safely applied without violating the strong guarantees of a Full ORAM.

Acknowledgment

We thank the anonymous reviewers for many helpful suggestions. This work was supported in parts by NSF grant CNS-1718834 and Intel. Chandrasekhar Nagarajan is currently affiliated with Micron and Ali Shafiee is currently affiliated with Samsung; this work was performed while they were graduate students at the University of Utah.

References

[1] S. Aga and S. Narayanasamy. 2017. InvisiMem: Smart Memory for Trusted Computing. In *International Symposium on Computer Architecture*.

[2] A. Awad, Y. Wang, D. Shands, and Y. Solihin. 2017. ObfusMem: A Low-Overhead Access Obfuscation for Trusted Memories. In *International Symposium on Computer Architecture*.

[3] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 8.

[4] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti. 2012. *USIMM: the Utah SImulated Memory Module*. Technical Report. University of Utah. UUCS-12-002.

[5] Claire Cain Miller. [n. d.]. Revelations of N.S.A. Spying Cost U.S. Tech Companies. https://tinyurl.com/y9syuvwg.

[6] Wind Company. 2007. Wind River Simics Full System Simulator. http://www.windriver.com/products/simics/

[7] C. Fletcher, L. Ren, A. Kwon, M. van Dijk, and S. Devadas. 2015. Freecursive ORAM: [Nearly] Free Recursion and Integrity Verification for Position-based Oblivious RAM. In *Proceedings of ASPLOS*.

[8] C. Fletcher, L. Ren, X. Yu, M. van Dijk, O. Khan, and S. Devadas. 2014. Suppressing the Oblivious RAM Timing Channel While Making Information Leakage and Program Efficiency Trade-Offs. In *HPCA*.

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

[10] O. Goldreich. 1987. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *Proceedings of STOC*.

[11] O. Goldreich and R. Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* (1996).

[12] V. Goyal and A. Jain. 2013. On Concurrently Secure Computation in the Multiple Ideal Query Model. In *Proceedings of EUROCRYPT*.

[13] S. Gueron. 2016. A Memory Encryption Engine Suitable for General Purpose Processors. In *Proceedings of IACR*.

[14] Zecheng He and Ruby B Lee. 2017. How secure is your cache against side-channel attacks?. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 341–353.

[15] Hi Tech Global. 2018. Hybrid Memory Cube Module. http://www.hitechglobal.com/Accessories/HybridMemoryCube-HMC.htm.

[16] M. Y. Hsiao. 1970. A Class of Optimal Minimum Odd-weight-column SEC-DED Codes. *IBM Journal of Research and Development* 14 (1970). Issue 4.

[17] Micron Technology Inc. 2006. DDR3 SDRAM Part MT41J256M8.

[18] M. Islam, M. Kuzu, and M. Kantarcioglu. 2012. Access Pattern Disclosure on Searchable Encryption: Ramification, Attack, and Mitigation. In *Proceedings of NDSS*.

[19] B. Jacob, S. W. Ng, and D. T. Wang. 2008. *Memory Systems - Cache, DRAM, Disk*. Elsevier.

[20] J. Jeddeloh and B. Keeth. 2012. Hybrid Memory Cube – New DRAM Architecture Increases Density and Performance. In *Symposium on VLSI Technology*.

[21] Jordan Robertson and Michael Riley. [n. d.]. The Big Hack: How China Used a Tiny Chip to Infiltrate U.S. Companies. https://tinyurl.com/ycywjdmo.

[22] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. https://spectreattack.com/spectre.pdf.

[23] T. S. Lehman, A. D. Hilton, and B. C. Lee. 2016. PoisonIvy: Safe Speculation for Secure Memory. In *Proceedings of MICRO*.

[24] S. Li, K. Chen, M. Y. Hsieh, N. Muralimanohar, C. D. Kersey, D. Chad, J. B. Brockman, A. F. Rodrigues, and N. P. Jouppi. 2011. System Implications of Memory Reliability in Exascale Computing. In *SC*.

[25] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. 2015. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[26] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowic, and D. Song. 2013. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *Proceedings of CCS*.

[27] J. T. Pawlowski. 2011. Hybrid memory cube (HMC). In *Hotchips*.

[28] Ling Ren, Christopher W Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. 2014. Ring ORAM: Closing the Gap Between Small and Large Client Storage Oblivious RAM. *IACR Cryptology ePrint Archive* (2014).

[29] L. Ren, X. Yu, C. Fletcher, M. van Dijk, and S. Devadas. 2013. Design Space Exploration and Optimization of Path Oblivious RAM in Secure Processors. In *Proceedings of ISCA*.

[30] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *Proceedings of the 16th ACM conference on Computer and Communications Security*. 199–212.

[31] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens. 2000. Memory Access Scheduling. In *Proceedings of ISCA*.

[32] Brian Rogers, Siddhartha Chhabra, Yan Solihin, and Milos Prvulovic. 2007. Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly. In *Proceedings of MICRO*.

[33] A. Shafiee, R. Balasubramonian, M. Tiwari, and F. Li. 2018. Secure DIMM: Moving ORAM Primitives Closer to Memory. In *Proceedings of HPCA*.

[34] E. Shi, T. Chan, E. Stefanov, and M. Li. 2011. Oblivious RAM with $O((logN)^3)$ Worst-Case Cost. In *Proceedings of ASIACRYPT*.

[35] A. Sodani. 2016. Knights Landing (KNL): 2nd Generation Intel Xeon Phi Processor. https://www.alcf.anl.gov/files/HC27.25.710-Knights-Landing-Sodani-Intel.pdf.

[36] E. Stefanov and E. Shi. 2013. ObliviStore: High Performance Oblivious Cloud Storage. In *Proceedings of IEEE S&P*.

[37] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. 2013. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Proceedings of CCS*.

[38] M. Taassori, A. Nag, K. Hodgson, A. Shafiee, and R. Balasubramonian. 2018. Memory: The Dominant Bottleneck in Genomic Workloads. In *Proceedings of AACBB Workshop, in conjunction with HPCA-24*.

[39] M. Taassori, A. Shafiee, and R. Balasubramonian. 2018. VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures. In *Proceedings of ASPLOS*.

[40] Rujia Wang, Youtao Zhang, and Jun Yang. 2018. D-ORAM: Path-ORAM Delegation for Low Execution Interference on Cloud Servers with Untrusted Memory. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 416–427.

[41] Yao Wang, Andrew Ferraiuolo, and G. Edward Suh. 2014. Timing Channel Protection for a Shared Memory Controller. In *HPCA*.

[42] A.C. Yao. 1986. How to Generate and Exchange Secrets. In *FOCS*.

[43] Xiangyao Yu, Syed Kamran Haider, Ling Ren, Christopher W. Fletcher, Albert Kwon, Marten van Dijk, and Srinivas Devadas. 2015. PrORAM: dynamic prefetcher for oblivious RAM. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*.

[44] Xian Zhang, Guangyu Sun, Chao Zhang, Weiqi Zhang, Yun Liang, Tao Wang, Yiran Chen, and Jia Di. 2015. Fork Path: Improving Efficiency of ORAM by Removing Redundant Memory Accesses. In *Proceedings of the 48th International Symposium on Microarchitecture*.

[45] X. Zhuang, T. Zhang, and S. Pande. 2004. HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus. In *Proceedings of ASPLOS*.