Improving Performance and Space Efficiency of Secure Memory with ORAM

by

Mehrnoosh Raoufi

Bachelor of Science, University of Tehran, 2017

Submitted to the Graduate Faculty of

the Department of Computer Science in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2022

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF COMPUTER SCIENCE

This dissertation was presented

by

Mehrnoosh Raoufi

It was defended on

May 25, 2022

and approved by

Dr. Youtao Zhang, Department of Computer Science

Dr. Jun Yang, Department of Electrical and Computer Engineering

Dr. Xulong Tang, Department of Computer Science

Dr. Stephen Lee, Department of Computer Science

Improving Performance and Space Efficiency of Secure Memory with ORAM

Mehrnoosh Raoufi, PhD

University of Pittsburgh, 2022

Modern computer systems widely adopt the detached-memory architecture, i.e., the processor chip integrates a memory controller on-chip and sends memory addresses and device commands in cleartext on memory buses. Studies have shown that, even if the user data may be secured with strong encryption and authentication schemes, it is possible to leak sensitive information from access patterns in memory address traces. To ensure high-level protection of user privacy, it is necessary to adopt expensive ORAM (Oblivious RAM) primitive that obfuscates memory requests from the user program. ORAM converts each off-chip memory request from user program to tens to hundreds of memory accesses. While several schemes have been proposed to mitigate the total number of memory accesses. ORAM remains a highly memory intensive primitive that leads to large memory bandwidth and space occupation and performance degradation.

In this thesis, we study the most popular ORAM implementations and their latest optimizations proposed in the literature. We perform extensive analyses to expose ORAM inefficiencies in terms of bandwidth, performance and space demand. We then propose a set of techniques to address these inefficiencies. In particular, we present schemes to improve the performance of ORAM by reducing its memory intensity. Moreover, we propose to reduce the space demand of ORAM to make it more desirable for wide adoption.

Table of Contents

List of Tables

List of Figures

Preface

1.0 Introduction

1.1 Problem Statement

Modern computer systems widely adopt the detached-memory architecture, i.e., the processor chip integrates a memory controller on-chip and sends memory addresses and device commands in cleartext on memory buses [20]. Studies have shown that, even if the user data may be secured with strong encryption and authentication schemes, e.g., AES encryption [12], Merckle tree authentication [15], it is possible to leak sensitive information from access patterns in memory address traces [44, 39]. To ensure high-level protection of user privacy, it is necessary to adopt expensive ORAM (Oblivious Memory) primitive that obfuscates memory requests from the user program [16, 17].

A variety of ORAM implementations have been proposed over the years. They all tried to make ORAM more practical for wide adoption. ORAM is an expensive primitive in terms of memory space, bandwidth, and performance. Different implementation favors one aspect to improve at the expense of others. In this thesis, we study the most popular ORAM implementations and their enhancements. We analyze their inefficiencies and propose to improve each of them with a set of techniques.

Path ORAM [31] is a popular ORAM implementation that organizes the user data to be protected in a tree structure and converts each user memory request to one or multiple tree path accesses which translates to tens to hundreds of memory accesses. Path ORAM requires $2\times$ as memory space as an unprotected baseline. Half of the memory space is filled with dummy blocks in Path ORAM to ensure paths always have enough space to accommodate a real block. Note that this is crucial because paths are chosen randomly and the protocol would fail if a path overflows.

Memory access overhead in traditional ORAM [16, 17] was O(N) (where N is the number of data blocks of the protected memory space) since they demanded a full memory scan. Whereas in Path ORAM the overhead is reduced to O(logN) since the data blocks are organized in a full binary tree. Nevertheless, Path ORAM remains a highly memory intensive

primitive. Several optimizations have been proposed on top of Path ORAM. Section 2.5 discusses these enhancements. To protect 2GB of data, the Path ORAM tree requires to allocate 4GB of memory space. With a typical setting, the path length of the tree will be 96. It means each user request that is a miss on LLC will be translated to 96 block reads followed by 96 block writes.

Since the high memory intensity has become the main obstacle that prevents Path ORAM from wide deployment, many schemes have been proposed to mitigate memory bandwidth usage and its impact. Maas *et al.* proposed to cache top tree levels on-chip to reduce the number of data blocks to access [24]. Nagarajan *et al.* proposed to create a smaller tree such that majority accesses can be satisfied by the smaller tree, which reduces the length of the tree and the number of blocks per node [25]. Zhang *et al.* proposed to exploit the dummy blocks of the same path to save shadow copies so that the processor can resume execution early [42]. Unfortunately, the memory intensity of Path ORAM remains high, which still incurs large performance degradation to the user applications.

Ren *et. al.* proposed to reduce the bandwidth and performance overhead of Path ORAM by an order of magnitude. They proposed to do so via dedicating more reserved space for the protected memory. As discussed before, space utilization is 50% in Path ORAM. Whereas in Ring ORAM, the space utilization is reduced to 21%. To protect 2GB of data, Ring ORAM needs to allocate 10 GB. With a typical setting, each memory request is instantly translated to 24 block reads. Which is much less than Path ORAM bandwidth demand. However, Ring ORAM requires some write path operations while not serving user requests. The maintenance operations cost on average 99 block accesses (read and write). Ring ORAM outperforms Path ORAM in terms of execution time and bandwidth consumption. Its space demand, however, remains substantially larger than Path ORAM.

The following section discusses our contribution in this thesis to improve each ORAM implementation. We propose to reduce the memory intensity of Path ORAM in order to improve its performance and bandwidth demand. In addition, we propose to reduce the space demand for Ring ORAM.

1.2 Contribution Overview

In this thesis, we study the two most popular ORAM implementations; Path ORAM, and Ring ORAM. We also analyze the optimizations proposed in the literature on top of each. We then propose a set of techniques to improve each and make them more practical.

We identify the memory intensity in Path ORAM in terms of its different path types. We study all path types closely and identify the source of inefficiency in them. We then develop three different schemes to reduce the intensity of each.

While Path ORAM [31] reduces the memory access overhead from O(N) in traditional ORAM [16, 17] to O(logN) (where N is the number of data blocks of the protected memory space), it remains a highly memory intensive primitive that consists of path accesses of three types.

- **$\mathbf{PT}_p$ path**. To determine the tree path to access, Path ORAM uses multiple levels of position map, a.k.a., PosMap, tables to map user addresses to path IDs. While an on-chip buffer can cache frequently used entries, Path ORAM still needs to generate many PosMap accesses.
- **$\mathbf{PT}_d$ path**. To access a requested data block after knowing its path ID, Path ORAM accesses all tree nodes on the path from the leaf node to the tree root. All data blocks in these nodes are accessed to ensure secure protection.
- **$\mathbf{PT}_m$ path**. To prevent timing channel attacks, Path ORAM needs to generate path accesses at a fixed rate, e.g., one path access per T cycles [14]. When it needs to generate a path access but there is no pending real request, Path ORAM constructs a dummy one to access a random tree path.

In this thesis, we propose IR-ORAM that proactively reduces the memory access intensity of each path type. **IR-ORAM consists of a set of three path-type-dependent schemes with a focus on intensity reduction, i.e., it reduces the number of each type of path accesses to improve the overall performance.** Our contributions are as follows.

- We propose *IR-Alloc*, a utilization-aware node size allocation strategy, to reduce the number of data blocks to access for each path, i.e., the intensity of all types of paths. *IR-*

Alloc exploits the observation that tree nodes at different levels exhibit significant space utilization difference. Here, the *space utilization* is defined as the portion of memory blocks that saves real data blocks (rather than dummy blocks).

In particular, the middle level nodes show low utilization such that we can reduce the number of blocks allocated to these tree nodes, which effectively reduces the number of data blocks in each path while incurring minimized impacts on memory space and ORAM operation.

- We propose *IR-Stash* to reduce the number of PosMap accesses, i.e., the intensity of PT_p paths. Our utilization study reveals that the tree top mostly serves as an overflow buffer of the on-chip stash. However, existing schemes on buffering the tree top on-chip lead to either large space overhead or unnecessary PosMap accesses. We therefore develop *IR-Stash* that places top tree levels in a set-associative sub-stash and maintains its tree structure using a small table. *IR-Stash* helps to eliminate unnecessary PosMap accesses at low overhead.
- We propose *IR-DWB* to reduce the number of dummy path accesses, i.e., the intensity of PT_m paths. *IR-DWB* converts dummy paths to write-back operations of dirty LRU (least-recently-used) entries in the LLC (last level cache), which minimizes LLC replacement overhead while introducing no memory contention as that in traditional eager writeback cache designs.
- We evaluate the proposed techniques and study their effectiveness in memory intensity reduction. Our experimental results show that IR-ORAM achieves on average 42% performance improvement over the state-of-the-art while effectively enforcing the original memory access obliviousness and thus the same level of security protection.

As discussed before, Ring ORAM [28] is an ORAM implementation that is much faster than Path ORAM, however, it has much lower utilization. Path ORAM requires 2× space of unprotected memory whereas Ring ORAM requires 5× as big. While Ring ORAM performance is desirable, its large space demand may prevent its wide adoption. That is why in this thesis we attempt to improve the space efficiency of Ring ORAM. While we do not fill the space gap between Path ORAM and Ring ORAM entirely, we reduce it significantly.

To improve Ring ORAM space efficiency, first, we have to find out the source of waste. To this end, we conduct a set of extensive analytical experiments to demonstrate memory utilization across the Ring ORAM tree. We demonstrate two types of memory waste in Ring ORAM. First, lingering of dead blocks which remain useless until the next bucket refresh. Second, the over-allocated dummy blocks that occupy space though never used.

To address dead blocks accumulation, we propose to reuse them during execution for allocation. To this end, we propose a new allocation scheme called remote allocation. Remote allocation allows us to eventually reduce the bucket size and hence the memory allocated to the Ring ORAM tree.

We propose to shrink bucket size for the buckets at levels with the largest fair share in the capacity. We introduce a trade-off between performance and space saving. However, the performance impact is relatively small.

In this thesis, we propose different techniques to improve the performance and space efficiency of different ORAM implementations. For each scheme we first discuss the motivational observations then we describe our designs. The rest of the thesis is organized as follows.

In Chapter 2, we discuss the background of ORAM and we elaborate on the basics and the details of how different ORAM implementations work. First, we describe the common basics for Tree-based ORAMs. Then, we discuss the operations specific to Path ORAM and Ring ORAM.

In Chapter 3, we discuss different path access types in Path ORAM and we analyze the memory intensity of each type. Then, we describe a set of techniques we proposed to address the intensity of each path type.

In Chapter 4, we discuss analytical experiments that reveal the source of space waste in Ring ORAM. We then explain how we will exploit these observations to reduce space demand in Ring ORAM toward making it more practical for adoption.

In Chapter 5, we discuss our proposal for future work. We discuss the motivation as well as the schemes we plan to implement.

2.0 Background

2.1 Memory Basics

DRAM is the most commonly used technology as the main memory in modern computing systems. At the top level, it is a printed circuit board (PCB) that has several chips attached to it. The chips work in lockstep and each provides a portion of data to be accessed.

DRAM has a hierarchical organization; at the highest level, it has one or more channels, on each channel one or more DIMMs exist. Each DIMM is comprised of two ranks. Each rank has several chips (8, 16, etc.) that work in lockstep and each provides a portion of data to be accessed.

Each chip in DRAM also has a hierarchy of memory arrays. Each memory array can be thought of as a grid of rows and columns. Each chip has several independent banks of memory arrays. Memory arrays inside the banks are referred to as subarrays. A subarray refers to a group of rows in the memory. At the intersection of each row and column, there exists a DRAM cell. Each cell exactly represents one bit of data.

DRAM stands for dynamic random-access memory. Each DRAM cell is a single pair of transistor-capacitor. It is called random access because any location can be accessed directly via an address without needing to touch the preceding locations. It is dynamic because the capacitor storing the charge is not perfect and it can lose charge over time, hence it has to be refreshed periodically.

2.2 Memory Security and Privacy

Data privacy can be provided by storing data in an encrypted fashion. It does not require any support from the memory side. The processor store the encrypted data and once it reads it from the memory decrypts it to proceed with the execution. In this way, an adversary cannot learn about the content of data being transmitted over the bus.

In DRAM, address mapping is a scheme that resolves a physical address into DRAM indices in terms of channel ID, rank ID, Bank ID, row ID, and column ID. Commodity DRAM uses a direct-attached memory architecture where the address is communicated plaintext on the memory bus. Therefore, one by snooping the memory bus can learn about the memory location that processor accesses. Address access patterns can potentially leak information about user programs to the outside attacker.

Recent research has shown that the address access pattern can be protected using a crypto primitive called Oblivious RAM (ORAM). ORAM obfuscates the access pattern by reshuffling and re-encrypting the data blocks after each memory access.

2.3 ORAM History

This section describes the origins of ORAM and how it has evolved over time. The concept of ORAM was initially formulated by Goldreich in 1987 [16]. The goal was to protect against software IP theft. While cryptographic techniques can be applied to keep the contents of the memory secret, they cannot hide the access pattern of the program execution. A naive solution to obfuscate the program control flow is to scan the entire memory for each access. However, it is extremely inefficient both in terms of performance and bandwidth. [16] proposed Square Root ORAM to achieve a better performance. In this design, for protecting N data blocks $\sqrt{N}$ dummy blocks are stored alongside the real data blocks in the memory. All blocks are encrypted and permuted.

2.4 Path ORAM Basics

Path ORAM is a crypto primitive that protects memory access patterns through obfuscating memory accesses to untrusted external memory [31]. As shown in Fig. 1, Path ORAM organizes the untrusted memory space as a binary tree, referred to as *ORAM tree*. An on-chip ORAM controller converts each user memory request to a path access to the

ORAM tree.

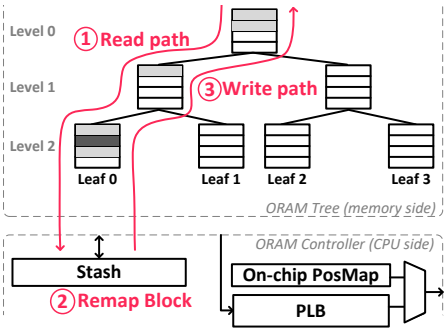


Figure 1: An overview of Path ORAM ($L = 3$, $Z = 4$).

The ORAM tree has L levels ranging from 0 (the root) to L-1 (the leaves). Each tree node, referred to as a *bucket*, has Z slots to save data blocks, e.g., cache lines in this thesis. The slots store either real data blocks or *dummy blocks*. Given all blocks are encrypted, the block types are indistinguishable.

The on-chip ORAM controller consists of control logic, stash, PosMap (position map table) and PLB (PosMap lookaside buffer). The PosMap is a lookup table that maps a block address *BlkAddr* in user address space to a unique *path ID* in the tree. The stash is a small fully associative on-chip buffer that temporarily keeps a number of data blocks (e.g., $100 - 200$ blocks) and their path IDs. Given PosMap could be big for a large user space, we may need multiple levels of PosMap and store these mapping entries in memory. We use PLB, a small on-chip buffer, to cache the frequently used PosMap entries such that we can reduce the number of ORAM accesses for PosMap entries.

ORAM operations. Path ORAM converts a memory request with *BlkAddr*=a to one or more path accesses to the ORAM tree. Each path access consist of the following phases.

1. *Stash, PosMap, and PLB access phase.* Before accessing the ORAM tree, the ORAM controller searches the block in the stash and, simultaneously, translates a to a path ID l using PLB/PosMap. The block is returned to the user application if it is found in the stash. Otherwise, the ORAM controller starts the read path phase if the corresponding a-to-l mapping is found in PLB, or fetches the mapping from memory.
2. *Path read phase.* Given a path ID l, the ORAM controller reads all the blocks on l from the memory. It decrypts and authenticates the fetched blocks, and discards the dummy

blocks and inserts the real blocks to the stash. This phase generates L×Z memory `read` accesses.

3. *Block remap phase.* After reading the path, the ORAM controller remaps block a to a random new path l'. It then updates the *PosMap* and the stash with the new map.
4. *Path write phase.* After returning the requested block to the user application, the ORAM controller writes data blocks except a back to path l. It searches the blocks in the stash and pushes the data blocks as low as possible in the ORAM tree. Dummy blocks are patched if not enough data blocks can be found. All blocks are encrypted and authenticated before being written to the memory. This phase generates L×Z memory `write` accesses.

Path ORAM may easily deplete the peak off-chip memory bandwidth [29], resulting in large performance degradation not only to itself but also to co-running applications [36]. In summary, the memory bandwidth consumption of Path ORAM has become the main obstacle that prevents its wide integration in modern computer systems.

2.5 Path ORAM Enhancements

Many schemes have been proposed to improve the performance of Path ORAM. We next discuss several closely related enhancements. Section 3.5 discusses more related work.

PosMap is sensitive metadata and thus requires secure protection. Path ORAM may recursively create a new ORAM tree for the PosMap and then a new level of PosMap. The last level of PosMap is saved in an on-chip buffer. For better access obliviousness and performance, Fletcher *et al.* proposed *Freecursive* to merge all these ORAM trees such that PosMap and data accesses are non-distinguishable [13]. In this thesis, we adopt Freecursive in the baseline. In our setting, we construct three levels of PosMap — PosMap_1 and PosMap_2 are merged in the main ORAM tree while PosMap_3 is saved completely in an on-chip buffer.

On average, Path ORAM/Freecursive introduces 8× slowdown over a non-secure execution in our setting. Given this large performance overhead, Path ORAM schemes are

currently applied to small applications or application kernels that are highly security sensitive.

Given the number of accessed memory blocks for each path is linear to the path length, Maas *et al.* proposed to buffer a few top levels on-chip [24]. They chose to save the tree top in the stash — they saved up to three levels without incurring considerable stash management overhead. Later studies [36, 25] proposed to buffer ten or more levels, which effectively reduces the memory bandwidth demand. Given the stash is fully associative, maintaining a huge stash with all tree top blocks becomes expensive. The tree top can be kept in a standalone buffer that maintains the tree structure [25].

To achieve high security, Fletcher *et al.* proposed to defend timing channel attacks by issuing path accesses at fixed rate and pattern [14]. A dummy access is inserted if there is no real user request pending. When a Path ORAM implementation has different types of path accesses, it becomes more complicated. For example, Nagarajan *et al.* proposed to construct an extra smaller ORAM tree so that there exists two path lengths [25]. To prevent potential timing channels, they issue path accesses using a fixed pattern, e.g., one main tree access after every four path accesses to the smaller tree. Dummy path accesses of either type are inserted as needed.

2.6 Ring ORAM Basics

Ring ORAM [28] was developed on top of Path ORAM [31]. Ring ORAM achieves performance improvement by reducing the memory bandwidth requirement for online accesses to $\frac{1}{Z'}$ of that in Path ORAM. An *online access* is referred to as the memory request from the user program.

Ring ORAM also organizes the to-be-protected memory as a binary tree. Assume we construct an ORAM tree with L levels and each bucket (i.e., tree node) has Z slots. Ring ORAM reserves S slots in each bucket, or $S \times (2^L - 1)$ for the entire tree, for holding dummy data only. The remaining $Z' \times (2^L - 1)$ slots may hold either real data blocks or dummy data (Z=Z'+S). Ring ORAM exploits around half of this space for real data blocks to facilitate

random path remapping, similar as that in Path ORAM.

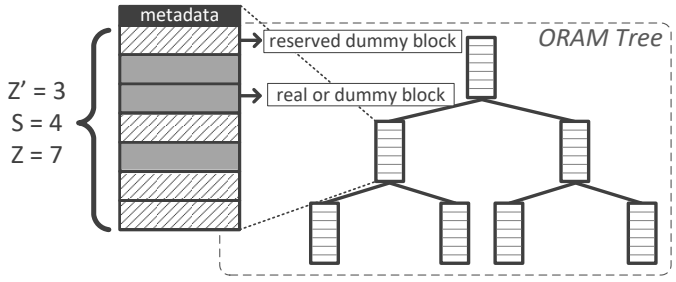


Figure 2: Ring ORAM tree organization ($L = 3$, $Z' = 3$, $S = 4$, and $Z = 7$).

Ring ORAM maintains a position map that maps each data block to a path id. It also includes a stash to buffer data blocks loaded from the memory, and optionally a tree top cache for performance improvement [27, 24]. Ring ORAM supports three types of operations as follows.

- ***ReadPath:*** this operation is to service *online accesses*. To access a block A, the ORAM controller first determines its mapped path l and then conducts a two-step access.

  (1) metadata access: The controller loads the metadata from a separate small tree for all buckets along l. It then identifies the location of A, i.e., a particular slot in one bucket, and then determines one valid dummy block from each of the other bucket along l. Metadata are updated/written back at the end of the Ring ORAM access.

  (2) block access: The controller reads one block from each bucket along the path. The block A is added to the stash while all other blocks are dummy blocks and thus discarded. The bucket location of each block is invalidated and the information gets updated in the metadata.

  This operation differs from its equivalent operation in Path ORAM in that it only reads one block per bucket.

- ***EvictPath:*** this operation is a background operation that gets triggered after every A online accesses. Each trigger chooses a path using reverse lexicographic order to reshuffle. It reads all remaining valid blocks from the buckets along the selected path, refills the buckets with loaded blocks as well as those in the stash, and write the new contents to the buckets in memory. The data are encrypted and authenticated and, as part of the

operation, the metadata are also updated. The role of this operation is to lower the stash occupancy and push the data blocks to levels close to tree leaves. It is similar to an access in Path ORAM but requires no specific block to access.

- ***EarlyReshuffle:*** this operation gets triggered for a particular bucket if it accumulates S *readPath* operations after the last *evictPath* reshuffled the bucket. To complete the operation, the controller reads the corresponding bucket into the stash, reshuffles and writes it back to the tree. Each bucket reshuffle includes Z' reads (from valid slots) and Z writes (to all slots).

Ring ORAM maintains metadata for each bucket, as listed in Table 3, to facilitate the protocol operation. When accessing a bucket with *readPath*, we increment its `count` and invalidate the corresponding slot. The `addr` and `ptr` fields determine at what slot each real block resides in the bucket.

Ren *et al.* explored the design space for choosing Z', S, Z, and A [28]. For a typical setting for secure processor setting, we use $Z' = 5$, $S = 7$, $Z = 12$, $A = 5$, as shown in [28].

Space utilization. For Ring ORAM, its space utilization is $Z'/(2*Z)$, or about 21% for the above typical setting.

2.7 Threat Model

In this thesis, we adopt the same threat model as Freecursive ORAM [13] and other existing ORAM studies [31, 29, 36, 25, 42]. We focus on preventing information leakage from the memory access traces of applications running on *not-fully-trustworthy* cloud servers. In such environments, the only trusted component (i.e., trusted computing base (TCB)) is the processor, i.e., while the processor can faithfully execute the user application, all other components, including the OS, the memory modules, and the memory buses, are not trustworthy. To ensure high level data security, we need to protect data secrecy, data integrity, and data privacy for the secure applications running on the servers.

We assume that data secrecy and data integrity are protected with hardware-assisted security enhancements [35, 19, 33, 15, 40]. As an example, the recently released Intel SGX

architecture saves encrypted user code and data in memory. They are decrypted when being brought into the processor [19]. A Merkle tree is built on the user data to prevent unauthorized changes [15]. Existing studies showed that the performance impacts of these enhancements are effectively mitigated with the trustworthy crypto hardware integrated in the processor.

In this thesis, we assume the attackers have the physical access to the servers so that they can trace and analyze all the memory accesses. The servers adopt the direct-attached memory architecture [20] such that, while the data on data buses are transmitted in ciphertext, the data on address buses and command buses are in cleartext. To protect the memory access patterns, the baseline adopts the traditional Path ORAM implementation [31] and its recent enhancements, as elaborated in the following sections.

3.0 IR-ORAM: Path Access Type Based Memory Intensity Reduction for Path-ORAM

3.1 Motivation

Since Path ORAM suffers mainly from frequent path accesses, we study the path types and the tree access patterns to reveal the opportunities for improvements.

3.1.1 The Types of Path Accesses

The preceding discussion reveals that there are three types of path accesses — PosMap paths, data paths, and dummy paths. They are referred to as $\mathbf{PT}_p$, $\mathbf{PT}_d$, and $\mathbf{PT}_m$ paths, respectively. To understand their importance, we conduct an experiment to evaluate the frequencies and summarize the results in Fig. 3. The experiment settings are in Section 3.3. Here, $\texttt{PT}_p\texttt{(Pos1)}$ indicates the memory accesses for fetching PosMap_1 entries, i.e., the mapping from the requested data's block addresses to their ORAM path IDs. $\texttt{PT}_p\texttt{(Pos2)}$ indicates the memory accesses for fetching PosMap_2 entries, i.e., the mapping from PosMap_1 entry's block addresses to their corresponding path IDs. Note that the entire PosMap_3 is saved on-chip. $\texttt{PT}_d$ indicates the memory accesses for fetching the paths that contain the requested data blocks. $\texttt{PT}_m$ indicates the dummy paths that are inserted due to lacking real requests because we need to defend timing channel attacks.

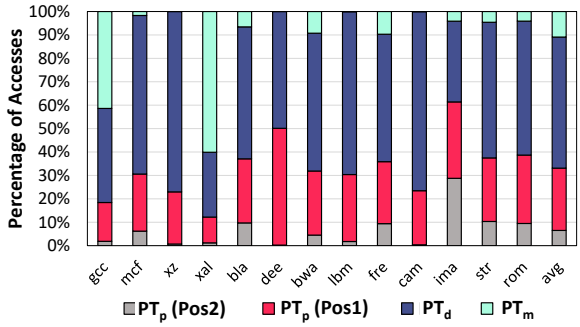


Figure 3: The distribution of different path accesses.

From the figure, (1) while $\texttt{PT}_d$ accounts for 56% of the total memory accesses, $\texttt{PT}_p$ is non-negligible — they are around 33% of the total. Of these accesses, $\texttt{PT}_p\texttt{(Pos1)}$ is around 4× of $\texttt{PT}_p\texttt{(Pos2)}$, indicating there are many more PosMap_1 misses than PosMap_2 misses to PLB. (2) $\texttt{PT}_m$ accounts for a large portion as well. In this experiment, we set the inter-path time interval T to be the same (T=1000 cycles) for all benchmarks. Setting the same T value achieves the highest security protection across different benchmark programs.

Alternatively, previous studies have shown that it might be possible to set the T value according to the memory intensity of the application. By setting a larger T value, fewer dummy paths may be inserted but real requests may have to wait longer before being serviced, which becomes problematic for bursty memory requests. In addition, an attacker may observe the T value to guess the memory intensity, introducing a potential information leakage channel, e.g., a covert channel.

Note, even though Path ORAM has three path types, its memory access obliviousness is securely enforced, i.e., an attacker cannot determine the type of a particular path access outside of the TCB — *the PATH ORAM controller keeps issuing path accesses at one per T cycles*. This principle is important for analyzing the memory access obliviousness.

To summarize, we conclude that Path ORAM contains three types of path accesses. To effectively reduce its memory bandwidth demand, we may develop schemes to address the memory intensity of every type.

3.1.2 The Utilization of Tree Nodes

To prevent stash full and protocol failure, Path ORAM keeps sufficient dummy entries in the ORAM tree — a typical implementation uses around 50% of the protected space to hold real data [29], i.e., we store around 4GB user data in a 8GB ORAM tree, with all the rest being dummy data blocks.

Given an ORAM tree consists of both (real) data blocks and dummy blocks, it is worthy to study the distribution of dummy blocks in the tree. Fig. 5.1 summarizes the results from an experiment for comparing the node space utilization (y-axis) at different levels (x-axis). We take snapshots at different execution times to illustrate the trend. Here, the *space*

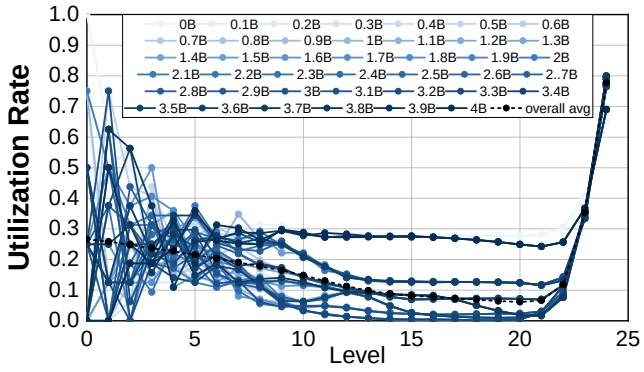


Figure 4: The space utilization at different tree levels.

utilization at a level is defined as the ratio of all useful data blocks to the total allocated memory slots at that level. In this experiment, we protect a 8GB memory space with 4GB user data. We have Z=4, L=25. The block size is 64B.

To initialize the ORAM tree, we clear the tree and access all data blocks once in a random order. For each block, we follow the Path ORAM baseline to remap and write the block to the tree. We create three levels of PosMap mapping tables accordingly. While there are 64 million data blocks in the ORAM tree, we run the memory trace for four billion path accesses, which is sufficiently long to show the *normal* access behavior, as shown in [31, 29]. Due to its excessive length, we use a mix of path accesses from the benchmarks (trace range [0B-3.7B]) and the randomly generated memory accesses (trace range (3.7B, 4B]). In the figure, "XB" indicates the snapshot after executing X billion path accesses, i.e., "0B" is the snapshot right after the initialization. Note that the average line indicates the overall average and not the average of taken snapshots.

Fig. 5.1 presents the snapshots at different execution points for the typical setting (as shown in the experiment section). While a similar study was reported in [31], they focused on choosing different bucket sizes. Instead, we make the following observations that are new in the literature.

- The top levels (level 0 to around level 9) exhibit large utilization fluctuations. For example, at level 3, the utilization ranges from 11% to 50% and there is no clear stable range

for the long execution at each level and across different levels. In general, the fluctuation tends to be more severe for the level that is closer to the tree root.

- The middle levels (level 10 to around level 21) exhibit two utilization ranges for different access patterns. For benchmark accesses that have patterns and data reuse (i.e., program memory traces), the utilization fluctuates at 20% and lower. For random accesses (i.e., synthesized memory traces), the utilization tends to be at around 30%. The utilization towards the end of the execution is around 30% because we place random traces at the end of the trace mix.
- The bottom levels (level 22 to 24) exhibit larger utilization than those of middle levels. In particular, the utilization of the last level tends to be around 70% for random accesses and 80% for patterned accesses. The utilization tends to grow towards that of the last level as they approach the last level.

When we fetch a tree path, each node contributes four data blocks indicating even memory bandwidth consumption across different levels. However, according to Fig. 5.1, we tend to get more dummy blocks from middle levels due to their low space utilization. In addition, a binary tree has significant capacity imbalance — the space of level l roughly equals to the space of all levels from 0 to l-1. While fetching top and middle levels consumes more memory bandwidth (e.g., 21 out of 25 levels), its space accounts for a small portion, e.g., top 21 levels occupy only 6% of the total space.

Fig. 5 compares the utilization of three different workloads. The results indicate that the utilization trend remains the same for individual workloads.

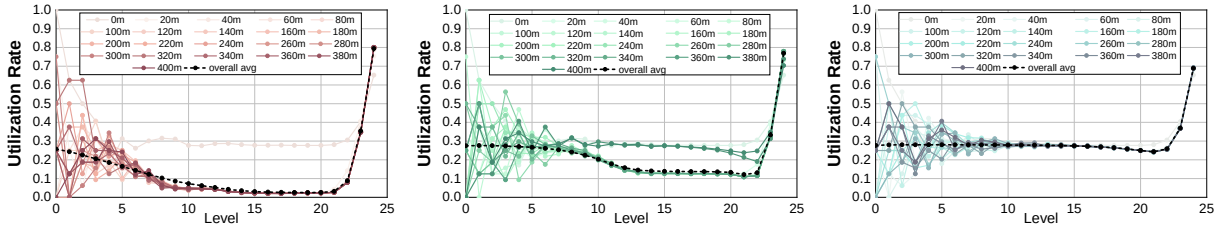


Figure 5: The space utilization behavior per benchmark; gcc (left), lbm (middle), and a random trace (right).

In summary, there exists a significant mismatch of node space utilization, memory band-

width, and space capacity across different tree levels.

3.1.3 The Block Migration Behavior

To understand the reason why an ORAM tree exhibits distinct utilization difference across different tree levels, we further study how a data block migrates in an ORAM tree.

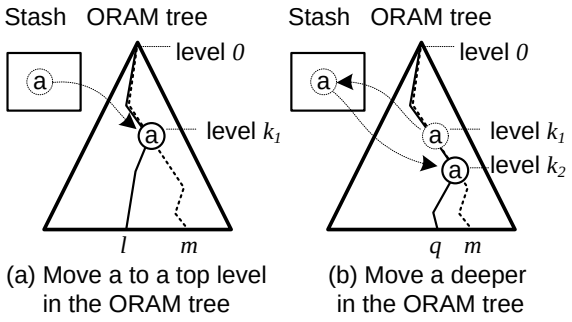


Figure 6: The migration behavior after a leaves the stash.

In Path ORAM, each path access has two memory phases — a read phase and a write phase. During the write phase, we choose data blocks from the read phase, *pre-existing* data blocks in stash, and/or dummy blocks and place them in the path. A pre-existing block is a block that was in the stash before the read path phase. The observation is, *if we pick up a pre-existing data block, we tend to place it to a top level.* For example, in Fig. 6(a), a is a pre-existing block. Assume we read path l and write blocks back to l, and we pick up a (mapped to path m) from the stash and write it to level k_1. We observe that k_1 tends to be a small value, i.e., k_1 is close to the root. This is because (1) any two paths overlap (because the root belongs to all paths); and (2) two random paths tend to have small path overlap. The latter is true because two paths overlap, e.g., at level 15, only if they belong to the same subtree at level 15. However, there are 32K different subtrees at this level, making the overlap possibility very low. In contrast, there are only 8 subtrees at level 3, it has higher possibility for two paths to overlap at level 3. Of course, due to limited capacity at top levels, a pre-existing data block may not get the chance to be moved to the ORAM tree.

We also observe that *data blocks fetched from the read path are likely to be flushed to the same or lower levels.* Fig. 6(b) illustrates how it works. Assume a (with path ID m) was

written to level k_1 (as in Fig. 6(a)). Now, we access another path q where path l and q overlap from level 0 (the root) to level k_2. we may not touch a if $k_1 > k_2$; and write to k_2 if $k_1 \leq k_2$. Due to the low utilization in middle levels, such write is likely to be successful.

We conduct an experiment to compare the node reuse at different levels. Fig. 7 summarizes the hit counts at different levels. From the figure, while top 10 levels account for less 0.01% of total ORAM space, the requested data blocks can be found in these levels for about 23% of total accesses.

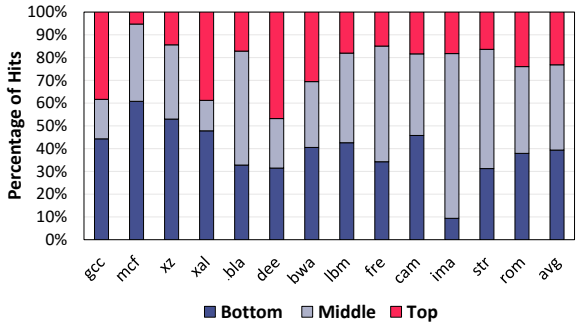


Figure 7: Nodes at top levels have high reuse possibility.

Therefore, *the top levels of an ORAM tree serve as an overflow buffer of the on-chip stash.* An accessed data block may be buffered temporarily in the stash and then written to the top levels of the tree. As the execution proceeds, a hot block is likely to be brought back to the stash to reuse while a cold block gradually sinks towards the leaf nodes of the tree.

3.2 The IR-ORAM Design

3.2.1 An Overview

In this thesis, we develop IR-ORAM to exploit our findings to reduce the memory intensity of each type of path accesses.

- We develop *IR-Alloc* to reduce the bucket sizes of middle level tree nodes. By exploiting the low utilization of middle level nodes, IR-Alloc reduces the number of data blocks to access for each ORAM path and thus the memory intensity of all three types of paths.

- We develop *IR-Stash* to architect a double-indexed set-associative sub-stash to adapt to large tree top caching. It reduces the number of PosMap accesses and thus the memory intensity of PT_p paths.
- We develop *IR-DWB* to convert dummy path accesses to useful early write-back operations, which reduces the memory intensity of PT_m paths.

3.2.2 IR-Alloc: a Utilization-aware Node Size Allocator for Reducing Intensity of PT_p/PT_d /PT_m paths

Traditionally, an ORAM tree uses one Z value, i.e., all buckets have the same number of data blocks. However, our study reveals that nodes at middle levels have considerable low utilization, indicating that 70% or more fetched data from these levels are dummy blocks and thus discarded after fetching. For this reason, it would be beneficial to shrink the bucket size at these levels. As an example, if we shrink the bucket to half of the original, we would read 50% fewer blocks for buckets at these levels.

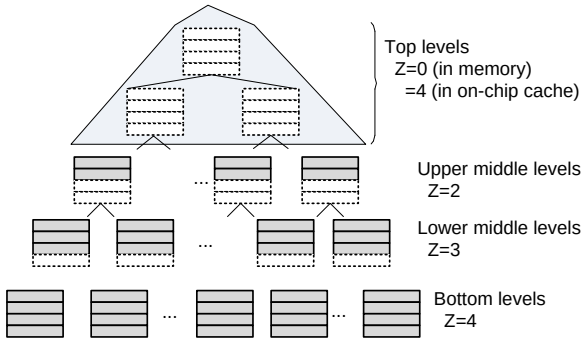


Figure 8: The design of IR-Alloc scheme.

Fig. 8 illustrates how *IR-Alloc* works. It adopts an allocation strategy with more than two Z values: Z=2, 3, and 4 for tree level ranges [10, 16], [17, 19], and [20, 24], respectively. [1] For completeness, we set Z=0 for memory allocation for tree level range [0, 9]. We will elaborate the details in the next section. With this allocation strategy, *IR-Alloc* only needs to access 43 data blocks (=10×0+7×2+3×3+5×4) during one read (or write) path phase.

[1]Here we use the math range representation, i.e., level range [a, b] indicates levels a, a+1, a+2, ..., b.

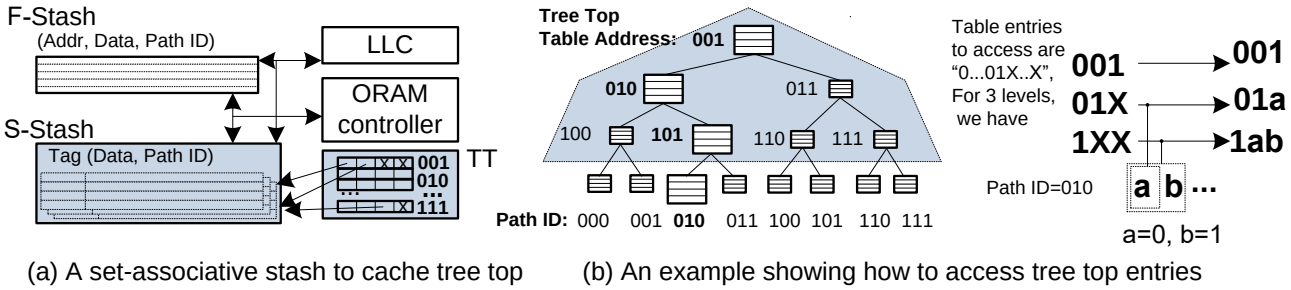
As a comparison, we need to access 60 and 100 blocks, respectively, for the Path ORAM designs with and without a 10-level top tree cache.

We next study the design issues in *IR-Alloc*. (1) One issue is the reduction of the total ORAM space as there are fewer block slots. This is negligible because the allocation in Fig. 8 only leads to around 0.9% space reduction. This is because most of the space of a binary tree is from lower levels. For example, the space from top 20 levels account for around 3.1% of the total space. (2) Another issue is that, while the middle levels satisfy the overall space demand, a particular tree node may not have the space to hold the chosen data block. Such a tree node may be able to hold that block if adopting the baseline Path ORAM implementation. This is in general not a problem as the ORAM controller would save the data block to a higher tree level, and eventually increase stash usage. As we experience more path accesses, the block still have the chance to sink towards to the leaf node.

The background eviction. The original Path ORAM places blocks that cannot move to the ORAM tree in the stash temporally [31] and faces protocol failure if the stash overflows, i.e., it has insufficient space to hold the blocks after reading a path. Ren *et al.* introduce background eviction, which effectively converts the correctness problem to a performance/overhead trade-off [29].

Since IR-Alloc reduces the number of empty slots on each path, fewer blocks may be moved to the ORAM tree during the write path phase, which increases the possibility of stash overflow. While it does not lead to security concern, too many background evictions may degrade performance significantly. We will analyze its security in Section 3.2.5 and study its performance impact in Section 3.3.

Choosing Z values. In this work, we adopt a greedy search algorithm to determine the Z values at different levels. This algorithm is based on empirical studies (as in the community and also shown in Fig. 3) that a random trace maximizes the utilization of stash entries as well as middle tree levels. Given an ORAM tree, we test run Path ORAM using random memory traces under two constraints: (1) the space reduction is within 1%; and (2) the increase of background evictions is within 15%. According to Fig. 5.1, random memory traces gives the worst case space utilization for middle levels, the focus of IR-Alloc. We observe that a 15% or less increase of background evictions for random traces exhibits



(a) A set-associative stash to cache tree top (b) An example showing how to access tree top entries

Figure 9: Exploit IR-Stash to effectively cache large tree top.

negligible increase of background evictions for benchmark traces. We start with setting Z=3 for level 19 (depending on the tree size) and Z=4 for all other levels, and gradually shrink lower levels Z values for finding a local maximal in performance improvement. The algorithm depends only on the ORAM configuration and, in particular, not on applications. Once the ORAM configuration is determined, we just go through the search process once and make the choice applicable for all deployed systems. Therefore, the search overhead is negligible in general. Note that we may want to run the search twice, once for IR-Alloc and once for IR-Alloc+IR-Stash as the background eviction rate may differ.

3.2.3 IR-Stash: a Double Indexed Stash Implementation for Reducing Intensity of PT_p Paths

Caching top tree levels is an effective way for reducing memory intensity. Maas *et al.* proposed to merge the top three levels to the stash and slightly increase the stash size to hold these blocks [24]. However, a major issue of this design is its scalability — the stash needs to be expanded to hold $(2^m\text{-}1)\times$Z more blocks if we cache top m levels. In addition, the stash needs to be fully associative. The stash may be accessed by the LLC using its block address (to determine if the requested block is in the stash), or by the ORAM controller using its path ID (to determine the blocks to expurge at write path phase). Therefore, it complicates the access if we make the stash a direct map (or a set associative) cache by indexing it using either the block address or the path ID. Unfortunately, integrating a fully associative buffer

is expensive. For example, if we cache top 12 tree levels, the enlarged stash has at least 16K entries. The corresponding die area is about that of a 4MB 8-way set associative LLC. Thus, it becomes less preferable when we need to move more top tree levels on-chip.

An alternative design is to buffer tree top in a dedicated on-chip cache and access them according to the tree structure, i.e., as they are in the memory [43, 36]. The stash can remain small (below 200 entries). In this design, the dedicated cache can only be accessed by the ORAM controller and thus is invisible to the LLC. Each ORAM path consists of two segments — top levels are in the buffer while the others are in the memory. The dedicated cache design harvests most of the benefits in caching. In particular, when reading a path, we search the buffer first. A buffer hit eliminates off-chip traffic and thus there is no need to remap.

A major issue with the dedicated cache design is the increased PosMap accesses. To determine if a data block is in the dedicated cache, the LLC needs to know the path ID of the block, which demands finding its Posmap entry. This access is necessary if the block is in the memory. However, if the block is in the cache, the PosMap access is a waste. Based on the discussion in Section 5.1, the top tree levels have a small size but a higher reuse possibility, which increases a non-negligible number of PosMap accesses. A PosMap access, if missed in PLB, results in a full path access, which significantly degrades the Path ORAM performance.

The IR-Stash Design. Fig. 9 illustrates our *IR-Stash* design. Intuitively, we include a large sub-stash for buffering the tree top entries and make it double-indexed to facilitate both LLC and ORAM accesses. The reason why IR-Stash can reduce the intensity of PT_p paths is that, if the tree top is indexed only by block addresses, a large number of PosMap accesses would be required to support ORAM path accesses. IR-Stash consists of two sub components.

(1) A small fully-associative stash *F-Stash* that has around 200 entries. *F-Stash* is the same as the traditional stash.

(2) A set-associative stash *S-Stash* that keeps blocks from the tree top. *S-Stash* is **double-indexed**. For the cache organization, it is indexed using the block address (in user space).

S-Stash is also indexed by a small pointer table TT, which helps to keep the tree structure of the blocks in *S-Stash*. Each entry in TT corresponds to a bucket and saves four pointers pointing to the data entries in *S-Stash* that save the data blocks contents and their path IDs. We skip the code with all zeros, assign the code "00..01" as the table address of the root node, and then continue the table address assignment level-by-level according to the tree structure.

In the example in Fig. 9(b), we cache top three levels and illustrate their table addresses.

We next describe how IR-Stash supports the accesses from both LLC and the ORAM controller. When LLC issues a request for a data block, it uses its block address to search both sub-stashes in parallel. Given *F-Stash* is small and fully associative, and *S-Stash* is set indexed by block addresses, the access is fast and incurs low access overhead. **A hit in either F-stash or S-Stash returns the block immediately and thus incurs no path access or path remapping.**

When the ORAM controller needs to access the tree top using a path ID, it follows the same Path ORAM protocol. The only difference is that the tree top is stored on-chip and thus uses TT table to identify the corresponding entries in *S-Stash*. To read a path, the ORAM controller reads the memory portion of the path and then the on-chip portion. For the on-chip portion, we need to access multiple buckets and access each bucket using its table address. By employing the coding strategy as discussed above, we can infer the tables addresses of these buckets. Note that maintaining TT is necessary because *S-Stash* is indexed by block address (in user space) which is different from the physical address of block location in the tree.

Assume the path to access is "$a_1a_2...a_i....$" ($1 \leq i \leq L$-1)" and we cache top t levels on-chip, we need to access t buckets and their table addresses are:

$$\overbrace{00...00}^{\textit{(t-1) zeros}}1,\quad \overbrace{00...00}^{\textit{(t-2) zeros}}1a_1,....,\quad \overbrace{0}^{\textit{1 zero}}1a_1...a_{t-2},\quad 1a_1...a_{t-1}$$

For each table entry, we follow its four pointers to fetch the block contents and their path IDs in *S-Stash*. To write a path, we follow the Path ORAM protocol to search *F-Stash* and choose the data blocks to write back at each level. To improve caching effectiveness and avoid address conflicts, *S-Stash* indexes the blocks using MD5 of their addresses. Our

experiments show that it evenly distributes the blocks.

When the ORAM controller fills in new blocks in the treetop buckets, we enter the chosen blocks in S-Stash and update the pointers accordingly. If a block from *F-Stash* cannot be moved to *S-Stash* because the target cache set is full, we skip picking this block for this round and get it flushed to *S-Stash* or a memory bucket at a later time.

At context switch, we flush the entries in F-Stash to the ORAM tree. Since the entries in S-Stash are cached tree bucket entries, they are encrypted, authenticated, and then written back to their corresponding memory locations. The TT table is then discarded. We rebuild the table to resume the execution.

3.2.4 IR-DWB: Converting Dummy Path Accesses for Reducing Intensity of $\mathbf{PT}_m$ paths

To mitigate the performance impact of dummy path accesses, we propose *IR-DWB* to convert them to useful memory accesses. *IR-DWB* converts dummy accesses into free early write-back of dirty blocks in LLC. Intuitively, *IR-DWB* searches for the dirty LLC entries that are likely to be written back in the near future, writes them to the memory of these entries, and marks these entries as clean so that it incurs low overhead when they are selected for replacement. Since dummy accesses are exploited for this matter, these early write-backs occur at no extra cost in terms of performance.

Fig. 10 illustrates how *IR-DWB* works. It consists of two main subtasks: (1) finding the appropriate dirty LLC entry; and (2) writing the dirty entry to memory. For the first subtask, we keep a register *Ptr* that points to the dirty LRU entry of one LLC cache set. We round-robin across all sets and search for the LRU entry when the LLC is idle. If the LRU entry of the current set is not dirty, we proceed to the next cache set. If the pointed entry is accessed and thus no longer an LRU entry, we clear *Ptr* (even if it is locked as discussed next) and proceed to the next cache set. If no entry can be found, we pause the search for 1000 cycles and restart from a random set. To implement a round-robin search, we used a small state-machine similar to autonomous eager writeback in [21] that is also adopted by [32, 38]. Alternatively, candidates can be chosen by maintaining a queue as in eager queue

scheme in [21]. The latter approach can be adopted in case hardware overhead is a tight constraint.

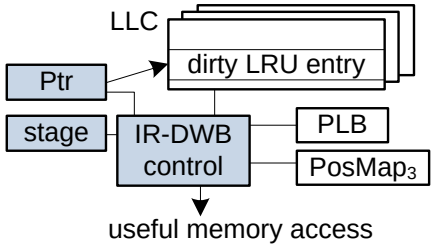


Figure 10: The design of IR-DWB scheme.

For the second subtask, we need up to three ORAM path accesses due to PosMap and data accesses, i.e., two ORAM path accesses for PosMap_1 and PosMap_2, respectively, and one path access for the dirty data block. For this purpose, we keep a register `Stage` to indicate if the corresponding LLC entry is ready to be written to the memory. We set `Stage`=3 if neither $PosMap_1$ or $PosMap_2$ mapping can be found in PLB; `Stage`=2 if $PosMap_2$ is a PLB hit while $PosMap_1$ is a miss; and `Stage`=1 if both can be found in PLB.

To defend timing channel attacks, Path ORAM issues path accesses at fixed rate and inserts dummy paths when there is no real request. IR-DWB optimizes the implementation as follows. When it is time to issue a dummy path access, *IR-DWB* checks if `Stage`=0 (indicating no LRU entry write-back is in progress). If `Stage`$\neq$0, *IR-DWB* continues the unfinished dirty entry flush; otherwise, it locks *Ptr* and proceeds to flush the dirty LRU entry pointed by *Ptr*. Depending on if we need PosMap accesses, we set `Stage`=3 for accessing $PosMap_2$ and `Stage`=2 for accessing $PosMap_1$; and `Stage`=1 if both PosMap entries are ready so that we can write the dirty data block. We decrement `Stage` after each path access and mark the corresponding LLC entry as clean when `Stage`=0. During this processing, if the dirty LLC entry pointed by *Ptr* is no longer a LRU entry, we abort the early eviction by setting `Stage`=0; or if the entry is chosen as a victim entry, we abort the early eviction by setting `Stage`=0 and perform the normal eviction instead.

From the discussion, for maximized benefit, IR-DWB requires three dummy path accesses to write the selected LLC entry back to the memory. In practice, we gain benefits if we have one or two dummy path accesses and thus can only partially service the request.

Delayed Block Remapping. IR-DWB currently works with the traditional LLC eviction strategy, i.e., a data block is remapped after its access; it is then placed in the LLC [31, 13, 41]. An LLC-evicted data block, if being dirty, becomes a new memory access. Alternatively, Nagarajan *et al.* [25] proposed a delayed data block remapping policy [25]. It works as follows. After accessing a data block, the ORAM controller discards its mapping, which eliminates the data block from the ORAM tree. The data block is added back to the ORAM tree when it is evicted from the LLC. Under this policy, the writeback block uses the ORAM-removed block in stash and thus shall not increase stash pressure. However, there is a limitation, i.e., it demands PosMap accesses at write-back time. Given the corresponding PosMap entry for servicing the initial access may not be in the PLB, it needs up to two extra full path accesses for PosMap entries. The LLC and memory are not inclusive under this policy.

Comparing the two data block remapping policies, the delayed remapping tends to introduce extra PosMap access overhead when most of evicted data blocks from LLC are clean. As shown in the experiment section, while the delayed remapping improves the overall baseline performance, it may slowdown the read intensive benchmarks by more than 50%.

To integrate IR-DWB with delay data block remapping, we may proactively conduct block remapping for LRU entries in LLC, and convert dummy paths to PosMap accesses to eliminate the overhead at write-back. We may need extra table for the generated remapping and thus leave it to future work.

Comparison. *IR-DWB* shares the similarity with *eager writeback* [21] for speeding up the traditional write-back LLC. Both schemes evict the dirty LLC entries before they are chosen for replacement. However, they differ as follows. (1) *eager writeback* tends to increase off-chip memory bandwidth demand. *IR-DWB* only exploits dummy path accesses. Since dummy path accesses consume the bandwidth anyway, *IR-DWB* does not introduce extra memory bandwidth demand. (2) *eager writeback* may degrade the program execution as an ongoing writeback request may block a user request that otherwise can be issued. *IR-DWB* does not hurt the current or the next request as a dummy path access needs to finish before the ORAM controller may issue another path access. Converting the dummy access does not degrade the execution. (3) one *eager writeback* can be serviced by one extra memory

access while one *IR-DWB* may need up to three dummy paths.

3.2.5 Security and Correctness Analysis

Security Guarantee. Path ORAM is a security primitive that protects user privacy through memory address obfuscation. It achieves memory obliviousness with two uniformity.

(1) **Path accesses are not distinguishable**. Even though there exists read or write user requests, and ORAM path accesses can be categorized as three types (data block access, dummy path access, and PosMap access), all path accesses look the same. An attacker outside of TCB cannot distinguish either the memory request type or the path access type.

Note, each path consists of one bucket access (or Z block accesses) to each tree level. Given the memory addresses are in cleartext, the tree access is non-oblivious, i.e., an attacker knows exactly when and what tree nodes are accessed.

(2) **Access intensity is not distinguishable**. By enforcing timing attack protection, the ORAM controller issues one path access every T cycles. An attacker outside of TCB cannot infer the path access type or application behavior. In particular, if timing attack protection is not enforced, the first path access of a burst of several path accesses is more likely to be a PosMap access.

In this section, we show that IR-ORAM enforces the above uniformity and thus the same level of security protection. While IR-Alloc reduces the number of blocks read from some tree levels, all paths are kept the same. Each individual path fetches the same number of block from a particular tree level so that we cannot distinguish the type of a particular path from the rest of all others. Similarly, eliminating access tree top nodes in IR-Stash and converting dummy paths to useful paths in IR-DWB keep the obliviousness of path accesses.

The non-uniformity introduced in IR-Alloc, e.g., we fetch two blocks from level 12 and four blocks from level 23, leaks no sensitive information. This is because memory addresses are in cleartext, and the block-to-tree-level mapping is public information in the original Path ORAM design. Since it is well known how the bucket sizes are adjusted, exposing node level non-uniformity has no security concerns. For the top tree cache design, we will not start

off-chip memory accesses until we know if the requested block is in the on-chip sub-stashes and prevent potential information leakage.

In summary, the ORAM paths in IR-ORAM are kept the same pattern and at the fixed rate. Thus, it ensures the same level of security protection as that in the original Path ORAM.

Correctness Guarantee. IR-ORAM does not introduce correctness issue either. IR-ORAM changes the way how the stash and the ORAM tree are constructed, which increases the possibility of stash overflow. In the basic Path ORAM implementation, the protocol fails if a stash overflow happens. Ren *et al.* introduced background eviction [29], which effectively converts the protocol correctness problem to a performance/overhead trade-off. In IR-ORAM, we enable background eviction if there are more blocks than a threshold after any write path phase. In summary, IR-ORAM does not introduce correctness issue to Path ORAM.

3.3 Experiment Methodology

To evaluate IR-ORAM, we used a trace-based simulator USIMM for cycle-accurate DRAM memory simulation [7]. This is similar to the setting that was adopted in recent Path ORAM studies [25, 36]. As shown in Table 1, we modeled a 4-issue OoO (out-of-order) 3.2GHz processor. There are two levels of data cache with the LLC (last level cache) being 8-way set associative 2MB. We modeled the ORAM tree following the typical setting [13, 42, 25] — we protect 8GB memory with 4GB user data. We use Z=4, L=25 for baseline.

To collect the trace for evaluation, we used Pin tool [23] on SPEC CPU2017 suite [1]. For each program, we skipped the warmup phase and then collected the trace that covers 2M L1 cache misses. We also picked a few traces obtained from PARSEC suite [5, 7]. Table 2 lists the L2 misses per kilo instruction (MPKI) for all the benchmarks we picked.

Table 1: System configuration.

Processor Configuration	
Processor Fetch Width/ ROB Size	4 / 128
Memory Channels	4
DRAM Clk Frequency	800 MHz
L1 D-cache	2-way 256KB
L2 cache (LLC)	8-way 2MB
ORAM Configuration	
Protected space and user data	8GB/4GB
ORAM tree levels	25
Bucket size/Block size	4 / 64B
Stash entries	200
Dedicated tree top cache	256KB (4K entries)
On-chip PLB / PosMap	64KB / 512KB

Table 2: Evaluated benchmarks.

Suite	Benchmark	read MPKI	write MPKI	Benchmark	read MPKI	write MPKI
SPEC	gcc	0.1	0.3	bwa	0.0	20.7
	mcf	19.5	0.1	lbm	0.0	45.3
	xz	24.9	29.6	cam	0.01	8.8
	xal	0.05	0.1	ima	0.3	2.9
	dee	0.0	5.7	rom	0.02	23.0
PARSEC	bla	2.6	0.4	fre	2.1	0.4
	str	2.7	0.5			

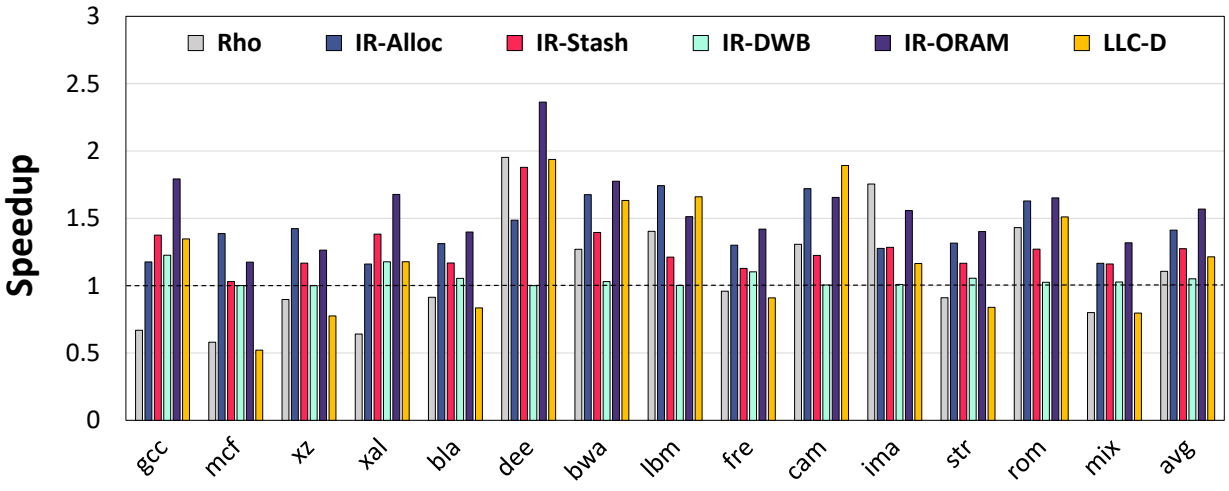


Figure 11: The performance comparison of different schemes.

3.4 Experimental Results

We implemented and compared the following schemes.

- `Baseline`: this is the traditional Path ORAM implementation [31] that adopts Freecursive [13] and has ten top tree levels cached in dedicated on-chip cache. It also adopts the subtree layout to improve row buffer hits and background eviction to prevent stash overflow [29].
- `Rho`: it is the ρ design [25] over `Baseline`. `Rho` implements a smaller tree and several other optimizations. We chose the best setting (L=19, Z=2) for the small tree, included other optimizations, and enforced the defense for timing channel attacks (we used 1:2, i.e., one main tree access per two accesses to the smaller tree).
- `IR-Alloc`: it implements IR-Alloc over `Baseline`. We set Z=1 for tree level range [10,15], and Z=2 for [16,18].
- `IR-Stash`: it implements IR-Stash over `Baseline`. For *S-Stash*, we tested different set associativities and choose 4-way set associative in this thesis.
- `IR-DWB`: it implements IR-DWB over `Baseline`.
- `IR-ORAM`: it integrates all three designs in our thesis. It is built on top of `Baseline`. (It sets Z to 2 and 3 for tree level ranges [10,16] and [17,19], respectively.)

- `LLC-D`: it adopts the delayed data block remapping policy [25] on top of `Baseline`.
- `IR-Stash+IR-Alloc` (LLC-D as baseline): it is `IR-Alloc` and `IR-Stash` on top of `LLC-D`.

3.4.1 Performance Comparison

Fig. 11 compares the performance of different schemes. The results are normalized to `Baseline`. *mix* bar indicates mix trace of 3 different benchmarks. From the figure, `Rho` achieves an average of 11% improvement. It exhibits a large degradation on *mcf*. This is due to the mechanism integrated for defending timing channel attacks, which inserts many dummy paths and offsets the benefit from accessing the smaller tree. This reduction may be mitigated if making the defense application-specific (currently we used 1:2 ratio).

For our schemes, `IR-Alloc` achieves on average 41% improvement over `Baseline`. The improvement comes mainly from the reduced number of data blocks to access for each path. Since we reduce the memory intensity of all path types, the improvements are stable across all benchmark programs. `IR-Stash` achieves on average 27% improvement over `Baseline`. Given both `Baseline` and `IR-Stash` cache top ten levels of the tree, the improvement comes mainly from the reduction of PosMap accesses. `IR-DWB` achieves on average 5% performance improvement. When there were more dummy path accesses, e.g., *gcc* in the figure, we found more opportunities for conversion, which helped to achieve higher improvement. For benchmarks having few dummy path accesses, e.g., *cam* and *dee*, `IR-DWB` was rarely activated, which led to close to zero performance impact.

When enabling all three proposed schemes, `IR-ORAM` achieves on average 57% improvement over `Baseline`, or 42% improvement over `Rho`.

`LLC-D` improves `Baseline` for most of the benchmarks due to their high dirty eviction rate. However, a read intensive benchmark like *mcf* experiences 1.9× slowdown. Fig. 12 illustrates the speedup of `IR-Stash+IR-Alloc` over a baseline that adopts `LLC-D`. Our two schemes can effectively improve a baseline with `LLC-D` adopted by 72% on average. We observe a high speedup of 1.63× for *mcf*. This is because, with `LLC-D` baseline, the number of hits in the tree top triples for this benchmark such that `IR-Stash` finds more opportunities

to reduce the number of PT_p path accesses.

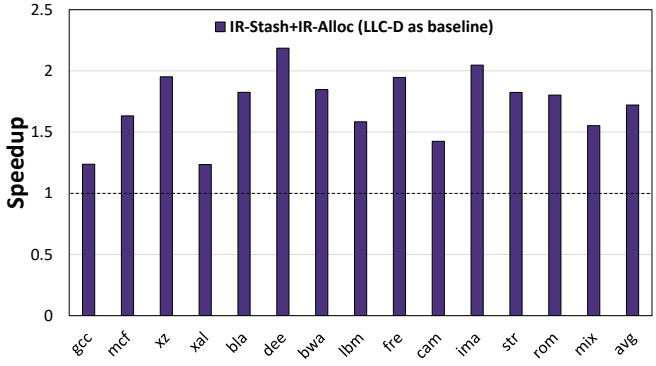


Figure 12: The performance comparison with LLC-D as baseline.

Since `IR-Stash` and `IR-Alloc` are orthogonal to timing channel protection feature, we also measured their speedup without having timing channel protection. Our results showed that `IR-Alloc` achieves slightly smaller speedup compared to when timing channel protection was enabled (40% vs 41% in Fig. 11). This is expected because with timing channel protection being enabled some background eviction accesses are reduced due to existence of inevitable dummy accesses.

By developing path type dependent techniques, `IR-ORAM` effectively reduces the memory intensity of the Path ORAM.

3.4.2 IR-Alloc Overflow

While `IR-Alloc` changes the Z values for two tree level ranges, the discussion in Section 3.2.2 indicates that this selection is not unique — we may choose different Z values for different tree level ranges, and all such selections may give good performance improvements.

To study the selection of Z values, we composed four configurations that set Z values for different tree level ranges as follows. PL indicates the number of data blocks we need to fetch per path. For all configurations, the memory shrinks below 1%. Of course, there are more configurations available.

- `IR-Alloc1:Z=2` for `L10`∼`16`, `Z=3` for `L17`∼`19` .... `PL=43`

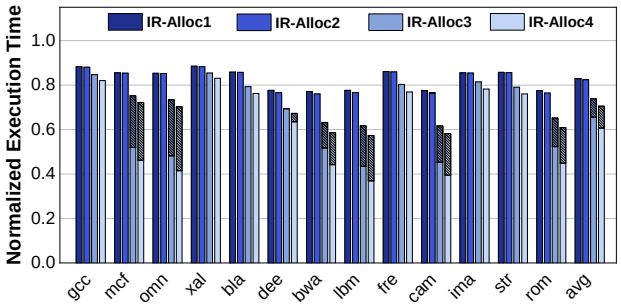


Figure 13: Design exploration of IR-Alloc scheme.

- `IR-Alloc2: Z=2` for `L10∼16`, `Z=2` for `L17∼18` .... `PL=42`
- `IR-Alloc3: Z=1` for `L10∼14`, `Z=2` for `L15∼18` .... `PL=37`
- `IR-Alloc4: Z=1` for `L10∼15`, `Z=2` for `L16∼18` .... `PL=36`

Fig. 13 compares the performance of different `IR-Alloc` configurations. The results are normalized to execution time of `Baseline`. For each bar, the shaded portion indicates the time spent on background eviction. `IR-Alloc4` is the same as `IR-Alloc` in Fig. 11. From the figure, in general, we tend to achieve larger performance improvement by reducing the number of data blocks to access per ORAM path. In addition, aggressively reducing the number of data blocks tends to incur large time in background eviction.

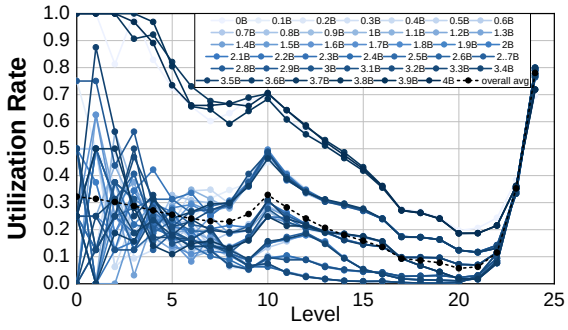


Figure 14: Level utilization with IR-Alloc.

To study background eviction, Fig. 14 shows the level utilization experiment similar to that in Section 5.1 for `IR-Alloc`. Snapshots were taken at different times of the execution using the same memory trace mix as in Fig. 5.1.

From the figure, we observed that, for memory accesses from benchmarks, the top and middle tree levels show high space utilization ratios than those before adopting `IR-Alloc`, i.e., the utilization ratios in Fig. 5.1. However, they are still low, meaning in general, the background eviction is still low.

For randomized traces, the utilization ratios are much higher, i.e., more than 50% utilization. In particular, we rarely found empty slots for tree nodes between level 0 and level 3. This indicates there is a high possibility of stash overflow. However, most benchmark programs have stable working sets and `IR-Alloc` achieves performance improvements over the Path ORAM implementation.

3.4.3 PosMap Reduction

We next investigated the effectiveness of `IR-Stash` in reducing position map accesses. Fig. 15 reports the normalized PosMap accesses of `IR-Stash` over `Baseline`. From the figure, `IR-Stash` dramatically reduces the number of PosMap accesses — on average, the number of PosMap accesses in `IR-Stash` are 49% of those in `Baseline`.

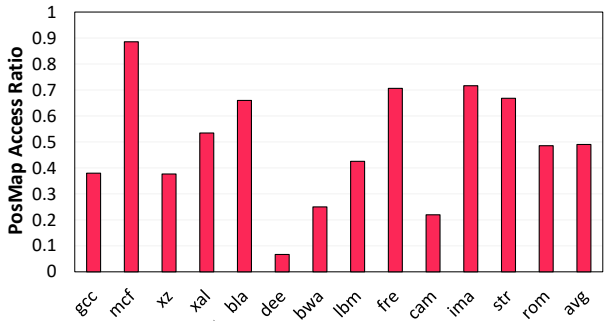


Figure 15: Comparing PosMap accesses with Baseline.

For benchmarks that have large reduction, e.g., 94% for *dee*, `IR-Stash` achieves large improvement of 87%. For the one with small reduction, e.g., *mcf*, it achieves low improvement.

3.4.4 Dummy Path Accesses

`IR-DWB` is designed to exploit dummy accesses in timing channel protection mode for early

write-backs. Fig. 11 shows its speedup. We also investigated its effectiveness in converting dummy accesses. Fig. 16 illustrates the percentage of each path access type. On average,

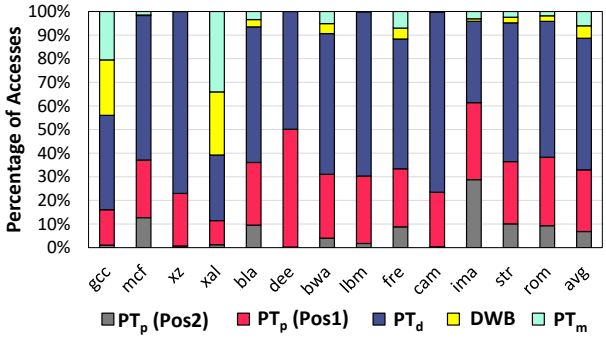


Figure 16: Access type distribution in IR-DWB.

`IR-DWB` reduces the percentage of dummy accesses from 11% to 6%.

3.4.5 Scalability Analysis

To evaluate the scalability of `IR-Alloc`, we used different sizes of protected memory, i.e., 2GB (L=24) and 8GB (L=26), and summarized the results in Fig. 17. For each configuration, we applied the Z finding algorithm accordingly to find the appropriate Z value at each level. We used the random traces as they set the performance lower bound while exhibiting high probability in background eviction. Fig. 17 compares the speedup of `IR-Alloc` over `Baseline` for different memory sizes. The x-axis indicates the size of user data which is half of the entire protected space. We conducted the experiment for 13 different random traces and reported the average speedup. The standard deviation of different speedup results is low (=0.0001) as random traces lack locality.

Impact of block size. When adopting larger blocks, e.g., by a cloud server with remote clients, the PosMap becomes smaller, which may be stored completely within TCB. This eliminates the potentials that `IR-ORAM` can explore from `IR-Stash`. However, the benefits from `IR-Alloc` remain untouched.

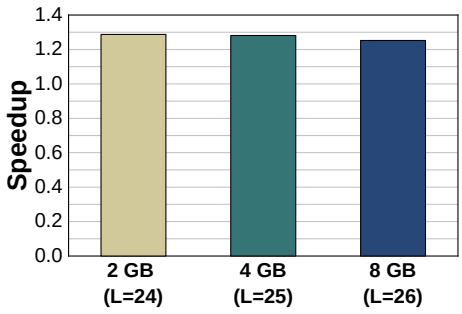


Figure 17: The scalability analysis of IR-Alloc.

3.4.6 Overheads

Space overhead. By reducing the bucket sizes of selected tree levels, `IR-Alloc` reduces the total memory space. However, this reduction is negligible, the different `IR-Alloc` configurations in Section 3.4.2 keep the space loss below 1%.

The `IR-Stash` design, comparing to the dedicated tree top cache design, keeps the tree structure in a small table, which saves $(2^{10} - 1) \times 4$ pointers and each pointer is of 12 bits. The total size is 6KB. In addition, it saves the tag array that the dedicated tree top cache may not need. This overhead is modest comparing to the enlarged stash design.

Energy overhead. The energy overheads comes from (1) the extra stash evictions introduced by `IR-Alloc`; (2) the extra table lookups in `IR-Stash`; and (3) the extra LLC/PLB accesses for finding the `IR-DWB` candidates. Given the energy consumption of Path ORAM comes mainly from memory accesses, the on-chip activities are negligible. For example, one access to 256KB cache is around 0.6nJ while an access to 1GB memory is about 40nJ [4]. The more extra stash evictions, the higher energy overhead the `IR-ORAM` may have. In our experiments, the number of extra stash evictions is low, incurring less than 5% of total energy consumption. Our energy saving in the memory systems is proportional to performance improvement, i.e., about 57% over `Baseline`.

3.5 Related Work

Ren *et al.* proposed Ring ORAM to reduce memory bandwidth at path access time with background tree reshuffle [28]. IR-ORAM adjusts the construction of the ORAM tree and thus is orthogonal and can be integrated to achieve better performance. Yu *et al.* proposed PrORAM to improve Path ORAM performance by constructing superblocks for prefetching [41]. Zhang *et al.* proposed to eliminate accessing overlapped portion of consecutive paths [43]. Wang *et al.* proposed to mitigate the interference of Path ORAM on co-running processes on the same server [36]. Path ORAM was also enhanced for its adoption for cloud services [30, 22].

By adopting secure memory architectures, memory access patterns on the buses can be effectively protected with hardware enhancements [2, 3]. Wang *et al.* proposed to achieve high level privacy protection by adopting Path ORAM for emerging BOB (buffer-on-board) memory architecture [37]. In addition to defending user privacy, hardware-assisted security enhancements are developed to mitigate the performance impact of data authentication [34].

3.6 Conclusions

In this thesis, we propose IR-ORAM, a Path ORAM optimization that consists of a set of techniques, to mitigate the high memory intensity of Path ORAM. In particular, we propose IR-Alloc to reduce the number of data blocks that we need to access for each tree path; IR-Stash to buffer top tree levels, which hybrids the extended stash and dedicated cache designs to achieve effective PosMap access reduction; IR-DWB to convert a significant portion of dummy path accesses to write back dirty LRU entries in LLC. On average, IR-ORAM achieves 42% performance improvement over the state-of-the-art while effectively enforcing the memory access obliviousness and the same level of security protection.

4.0 AB-ORAM: Constructing Adjustable Buckets for Space Reduction in Ring ORAM

Ring ORAM (Oblivious RAM) is a secure primitive that mitigates the large performance degradation of ORAM through reduced *online* memory bandwidth demand, i.e., the number of memory accesses at servicing a real memory request. Ring ORAM requires 4× or more of the protected data space to enable the optimization and thus presents high capacity pressure on modern memory systems. While recent studies strive to reduce its space consumption through bucket compaction, the large space consumption remains a major design challenge for Ring ORAM.

In this paper, we propose AB-ORAM to reduce the space capacity demand in Ring ORAM. AB-ORAM identifies two inefficient use of memory space in Ring ORAM: (i) accessed blocks hold useless data until the next reshuffle operation; and (ii) large buckets provide a diminishing performance benefit for tree levels close to the leaves. AB-ORAM then proposes two schemes to exploit the optimization opportunities, respectively. Specifically, it reclaims accessed blocks early by allocating them to buckets that need a reshuffle; and shrinks the bucket size for tree level close to the leaves for a better space/performance trade-off. We evaluate the proposed AB-ORAM design and compare it to the state-of-the-art. Our results show that AB-ORAM achieves an average of 36% space reduction over the state-of-the-art while introducing very low performance overhead.

Ring ORAM is a recently proposed secure primitive for mitigating the large performance degradation of ORAM [28]. Ring ORAM is built on top of Path ORAM [31], an ORAM primitive that organizes data blocks in a binary tree structure and converts each user memory request to two path accesses in the tree, which incurs large performance degradation, i.e., O(logN) complexity where N is the number of to-be-protected data blocks. Ring ORAM optimizes the protocol by differentiating two types of accesses: online and offline accesses. The former refers to those servicing the real user requests while the latter refers to those for protocol maintenance. While Ring ORAM has the same overall complexity as that of Path ORAM, i.e., O(logN), it fetches one data block from each tree bucket for online accesses,

representing $\frac{1}{Z}$ memory bandwidth requirement over Path ORAM, where Z is the number of data blocks in each tree node. With special hardware support, the memory bandwidth requirement for online accesses can be further reduced to O(1).

A major concern of Ring ORAM is its low space utilization. Path ORAM usually needs to double the memory space such that there are sufficient empty slots spreading across the ORAM tree and it has low possibility to remap a data block to a path with no empty slot. This leads to 50% **space utilization** [1]. Ring ORAM has even lower space utilization as it allocates more dummy blocks. For a typical setting [28], a 12-entry tree bucket keeps (1) five blocks for block remapping. On average, there are 2.5 block holding real data while the rest holds dummy data; and (2) seven dummy blocks for Ring ORAM protocol operations. This represents a 2.5/12= 21% space utilization. Given memory resource is precious for modern computer systems, the low space utilization tends to introduce large performance and energy consumption overheads. Cao *et. al.* addressed the space utilization issue with *bucket compaction (CB)*, a design that shrinks the bucket size and utilizes a portion of real blocks as reserved dummies when needed [6]. CB prevents the stash overflow possibility with more frequent background eviction, a performance/overhead trade-off optimization proposed for Path ORAM [29]. Unfortunately, space utilization with bucket compaction may be improved to 31%, which remains a major design obstacle for Ring ORAM adoption.

In this paper, we propose AB-ORAM to address the space inefficiency of Ring ORAM implementation. The contributions of AB-ORAM are summarized as follows.

- AB-ORAM exploits two inefficient use of memory space in Ring ORAM: (i) a data block becomes a *dead block* after its first access, and holds useless data till the next bucket reshuffle or path eviction; (ii) larger buckets that contain more dummy blocks help to support more path accesses till the next expensive bucket reshuffle or path eviction. However, for tree levels close to the leaves, its performance benefit diminishes fast while space demand increases dramatically.
- AB-ORAM addresses the inefficient use of memory space with optimized bucket allocation. AB-ORAM dynamically tracks dead blocks and adaptively allocates them to buckets

[1] In this paper, the space utilization is defined as the size of the real user data over the size of the ORAM tree.

that demand reshuffle, i.e., those due to bucket reshuffle or path eviction operations. This helps to reclaim dead blocks early and thus reduces the overall space demand.

By exploiting the space/performance trade-off at different levels, AB-ORAM adopts a statically fixed but non-uniform bucket space allocation strategy. It decreases the number of dummy blocks for the levels close to the leaves, which significantly reduces space demand at the cost of slightly increased bucket reshuffle frequency.

- We evaluate the proposed AB-ORAM design and compare it to the state-of-the-art. Our results show that AB-ORAM achieves an average of 36% space reduction over the state-of-the-art while introducing very low performance overhead.

4.1 Background

4.1.1 Attack Model

We use the same attack model as those in prior ORAM studies [31, 28]. Our discussion is based on a standalone secure processor while the design is applicable to cloud setting with secure server and remote clients. For the standalone secure processor, we assume that only the processor can be fully trustworthy, i.e., the trusted computing base (TCB) includes the processor only. The program code and data are stored in ciphertext in memory. An on-chip secure engine encrypts data before writing to memory, and decrypts after fetching from memory. The data are also authenticated to ensure data integrity. Prior studies have demonstrated that hardware-assisted security enhancements can effectively reduce the encryption and authentication overheads [35, 33, 15].

To prevent memory traces from leaking sensitive information, the baseline configuration adopts Ring ORAM. As we discussed, while Ring ORAM and Path ORAM share the same protocol complexity, Ring ORAM reduces the memory bandwidth requirement for online accesses and thus have better performance.

4.1.2 Path ORAM Basics

Given Ring ORAM is built on top of Path ORAM. We next briefly discuss how Path ORAM works. More details can be found in [31]. Path ORAM achieves O(logN) protocol complexity by organizing the to-be-protected memory space as a binary tree. Each tree node, referred to as a bucket, contains Z' data blocks while each data block is typically of cacheline size. Path ORAM includes an on-chip ORAM controller that contains a *stash* and a *position map*. The stash buffers blocks read from the tree while the position map holds frequently used mapping between data blocks and tree path IDs. Given a memory request for address A from the user program, Path ORAM translates A to its obfuscated path ID l, and services the request with two path accesses of l: read path phase and write path phase. Path ORAM accesses $L \times Z'$ blocks in each phase, where L is number of ORAM tree levels.

Path ORAM access has three types of operations. Assume we are to access block A.

① **read path:** The ORAM controller first looks up the position map to identify the path l on which block A resides, and then reads all the blocks on l from the memory. It decrypts and authenticates the fetched blocks. It keeps the real blocks in the stash and discards the dummy blocks.

② **block remap:** After the read path, block A is present in the stash. The ORAM controller remaps it to a random new path. It then updates the position map with this new mapping. Block A is then sent to the user program.

③ **write path:** The ORAM controller writes data blocks except A back to l. It searches the entire stash and starts the write-back from the leaf level to the root. Dummy blocks might be written to a tree node if there are not enough blocks found for that node. Note that all blocks are encrypted and authenticated prior to write-back.

The protocol fails if a data block is mapped to a path that contains no empty slot. To prevent protocol failures, Path ORAM uses only half of the slots to store real data blocks. Thus, the space utilization for Path ORAM is 50% [31, 29, 28].

	Metadata Field	AB-ORAM (bit)	Ring ORAM (bit)	Function
Block-related	`count`	$1 \times log(S)$	$1 \times log(S)$	Number of times the bucket has been touched since the last refresh
	`addr`	$Z' \times log(N_{Block})$	$Z' \times log(N_{Block})$	Address for each real block
	`label`	$Z' \times (L+1)$	$Z' \times (L+1)$	The path label of the each real block
	`ptr`	$Z' \times log(Z)$	$Z' \times log(Z)$	Offset in the bucket for each real block
	`valid`	$Z \times 1$	$Z \times 1$	Indicates whether the corresponding block is valid
	`remote`	$R \times 1$	-	Indicates whether the corresponding block is located at a remote address
	`remoteAddr`	$R \times log(N_{Bucket})$	-	Address of the bucket in which the corresponding block is remotely allocated
	`remoteInd`	$R \times log(Z)$	-	Offset in the bucket for each remote block
	`dynamicS`	$log(S)$	-	The current S value of the bucket (based on the last allocation)
Slot-related	`status`	$Z \times 2$	-	Indicates the slot status (`REFRESHED, ALLOCATED, DEAD`)

Table 3: Organization of bucket metadata in Ring ORAM and AB-ORAM.

4.1.3 Ring ORAM Basics

Ring ORAM [28] was developed on top of Path ORAM [31]. Ring ORAM achieves performance improvement by reducing the memory bandwidth requirement for online accesses to $\frac{1}{Z'}$ of that in Path ORAM. An *online access* is referred to as the memory request from the user program.

Ring ORAM also organizes the to-be-protected memory as a binary tree. Assume we construct an ORAM tree with L levels and each bucket (i.e., tree node) has Z slots. Ring ORAM reserves S slots in each bucket, or $S \times (2^L - 1)$ for the entire tree, for holding dummy data blocks only. The remaining $Z' \times (2^L - 1)$ slots may hold either real data blocks or dummy data blocks (Z=Z'+S). Ring ORAM exploits around half of this space for real data blocks to facilitate random path remapping, similar as that in Path ORAM. Figure 18 depicts Ring ORAM tree organization.

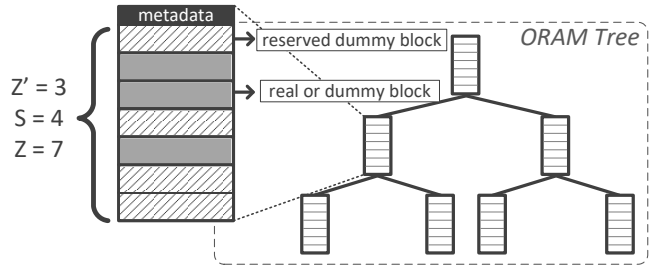


Figure 18: Ring ORAM tree organization ($L = 3$, $Z' = 3$, $S = 4$, and $Z = 7$).

Ring ORAM maintains a position map that maps each data block to a path ID. It also

includes a stash to buffer data blocks loaded from the memory, and optionally a treetop cache for performance improvement [27, 24]. Ring ORAM supports three types of operations as follows.

- ***ReadPath:*** this operation is to service *online accesses*. To access a block A, the ORAM controller first determines its mapped path l and then conducts a two-step access.

  (1) metadata access: The controller loads the metadata from a separate small tree for all buckets along l. It then identifies the location of A, i.e., a particular slot in one bucket, and then determines one valid dummy block from each of the other buckets along l. Metadata is updated/written back at the end of the Ring ORAM access.

  (2) block access: The controller reads one block from each bucket along the path. The block A is added to the stash while all other blocks are dummy blocks and thus discarded. The bucket location of each block is invalidated and the information gets updated in the metadata.

  This operation differs from its equivalent operation in Path ORAM in that it only reads one block per bucket, thereby reducing the memory bandwidth requirement compared to Path ORAM.
- ***EvictPath:*** this operation is a background operation that gets triggered after every A online accesses. Each trigger chooses a path using reverse lexicographic order to reshuffle. It i) reads all remaining valid blocks from the buckets along the selected path, ii) refills the buckets with loaded blocks as well as those in the stash, and iii) write the new contents to the buckets in memory. The data are encrypted and authenticated and, as part of the operation, the metadata are also updated. The role of this operation is to lower the stash occupancy and push the data blocks to levels close to tree leaves. It is similar to an access in Path ORAM but requires no specific block to access.
- ***EarlyReshuffle:*** this operation gets triggered for a particular bucket if it accumulates S *readPath* operations after the last *evictPath* reshuffled the bucket. To complete the operation, the controller reads the corresponding bucket into the stash, reshuffles and writes it back to the tree. Each bucket reshuffle includes Z' reads (from valid slots) and Z writes (to all slots).

Ring ORAM maintains metadata for each bucket, as listed in Table 3, to facilitate the protocol operation. When accessing a bucket with *readPath*, we increment its `count` and invalidate the corresponding slot. The `addr` and `ptr` fields determine at what slot each real block resides in the bucket.

Ren *et al.* explored the design space for choosing Z', S, Z, and A [28]. For a typical setting for secure processor setting, we use $Z' = 5$, $S = 7$, $Z = 12$, $A = 5$, as shown in [28].

Space utilization. For Ring ORAM, its space utilization is $Z'/(2*Z)$, or about 21% for the above typical setting.

4.1.4 Ring ORAM with Bucket Compaction

Cao *et. al.* proposed to shrink the bucket size while keeping Z' and S parameter intact by introducing the concept of overlap [6]. In this scheme, the bucket size is Z, and Z' blocks are dedicated to real blocks. Then, S can have a value of $Z - Z' + Y$, where Y is the number of blocks for overlap. In this way, when all dummy reserved blocks of a bucket are used, a block form the real blocks portion can be returned to the processor. That block is called a green block and may be either dummy or real. In case it is real, it has to remain in the stash. This can increase the chance of stash overflow. To address this issue, they proposed to generate dummy accesses if the stash occupancy reaches a threshold. Thus, dummy insertion continues until *evictPath* operation frees up the stash below the threshold. If we consider the typical setting of Ring ORAM, $Z' = 5$, $S = 7$, $Z = 12$ as a baseline, by applying compact bucket scheme with $Y = 4$, the allocation will be $Z = 8$, $Z' = 5$ and $S = 3$. In this paper, we built our design on top of this state-of-the-art.

4.2 Motivation

Ring ORAM, while achieving significant performance improvements over Path ORAM, exhibits low space utilization of 21% for the typical setting as discussed in Section 4.1. This low utilization is one of the main obstacles that prevent Ring ORAM form wide adoption.

We made two key observations regarding the space waste in Ring ORAM. In the following, we discuss these observations in detail and how we can exploit them to improve the space efficiency of Ring ORAM. The first observation is the existence of dead blocks in the ORAM tree. A dead block is the block invalidated upon a *readPath* access and is waiting to get refreshed by being rewritten. In the meantime, this block remains dead and useless. The second observation is that there exists a space/performance trade-off based on the bucket size and, for the levels close to the leaves, S value plays a more important role in space rather than the performance.

4.2.1 Studying Dead Blocks

A bucket slot in Ring ORAM tree can be accessed at most once between any two reshuffles. The Ring ORAM controller, while marking the slot as *invalid* after the access, does not reclaim the space until a later *evictPath* or *earlyReshuffle* operation that stores the newly shuffled, encrypted, and authenticated data. Given *evictPath* adopts reverse lexicographic order [28] and *earlyReshuffle* is on-demand, the duration of the slot being in *invalid* state can be relatively long, which lowers down the memory efficiency. In this paper, all invalid bucket slots are referred to as dead blocks. In this section, we study the total number of dead blocks and their lifetime.

Capacity. To study the capacity of dead blocks, we conduct an experiment to track the total number of dead blocks and summarize the results in Figure 19. The settings are in Section 4.5. The X-axis shows the program execution in the total number of online accesses. The Y-axis shows the snapshot of the total number of dead blocks. From the figure, the number of dead blocks increases quickly at the beginning of the execution and stabilizes after 30M online accesses.

Dead blocks are generated from online accesses, which have a stable rate, i.e., every *readPath* generates L dead blocks, where L is the tree height. The elimination of dead blocks, i.e., reclaiming invalid blocks through *evictPath* and *earlyReshuffle* operations, exhibits low rate at the beginning of the execution, and then stabilize for the rest of program execution. At the beginning of the execution, there are few dead blocks so that a few dead blocks can

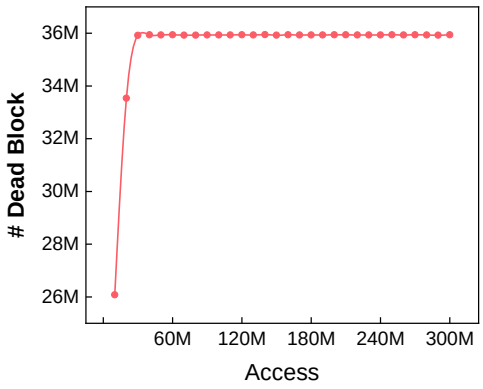


Figure 19: Dead blocks in the Ring ORAM over time.

be found from the selected path. With more online accesses, the dead blocks spread across all paths such that picking up any path get around L dead blocks to reclaim. This helps to stabilize the total number dead blocks.

We use a 24-level ORAM tree for Figure 19. The dead blocks account for around 18% (=36M/(12×(2^{24}-1)) of the total ORAM space. That is, around 18% of the allocated space is wasted at any time after entering the stable execution stage.

In Figure 20, the bars show the number of existing dead blocks for each level in Ring ORAM tree after running 400 million traces and the line indicates the number of buckets per level. As shown in the figure, the last level contains 17.9 million dead blocks. Given that the last level has about 8 million buckets, on average, there will be 2.1 dead blocks per bucket.

Lifetime. We next study the lifetime of dead blocks. The lifetime of a dead block is defined as the duration of the block being in invalid state.

We conduct an experiment to evaluate the lifetime of dead blocks at each ORAM tree level and summarize the results in Figure 21. The X-axis indicates the tree levels while the Y-axis indicates the lifetime of a dead block in terms of the number of online accesses. We report the minimum, the average, and the maximum lifetime of all dead blocks at each level. We skipped the warm-up phase, i.e., the first 30 million accesses, and used the accesses for the following 400 million accesses for each benchmark. We iteratively run a benchmark if its

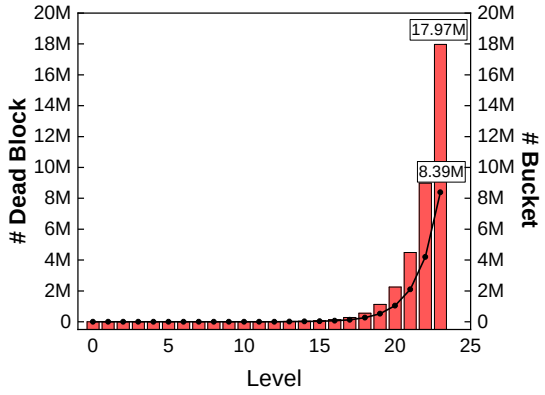


Figure 20: Dead blocks across the levels.

memory trace is shorter than 400 million. The three lines are the average of all benchmarks.

From the figure, dead blocks in buckets above level 18 (i.e., closer to the root bucket), tend to have a lifetime close to zero, indicating most of these dead blocks get reclaimed in a very short period of time. However, for dead blocks close to leaves, the average lifetime is large, indicating that dead blocks at these levels tend to be *invalid* for a long duration.

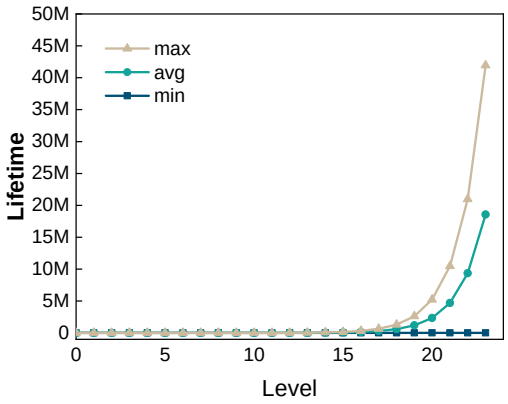


Figure 21: Dead blocks lifetime across tree levels.

Reclaiming opportunity. The existence of dead block is the major source of space waste that we target to address in this work. To this end, we propose AB-ORAM to significantly increase space efficiency without degrading the performance. Specifically, to mitigate space waste, AB-ORAM reduces the initial allocation of reserved dummy blocks for each bucket.

Then, to mitigate the potential performance loss caused by reduced initial dummy blocks, AB-ORAM attempts to increase the S value for each bucket dynamically. In particular, AB-ORAM explores and reuses the existing dead blocks in the ORAM tree.

4.2.2 Studying Space/Performance Trade-off

In this section, we discuss the intrinsic trade-off between space and performance in Ring ORAM. As discussed before, there are S dummy blocks allocated for each bucket in Ring ORAM as reserved dummy blocks. During the *readPath* operation, only one bucket returns a real block, and all other buckets along with the path return dummy blocks. Since each bucket has S reserved dummy blocks, if a bucket is touched S times, potentially it could run out of dummy blocks, hence, it has to be reshuffled. Thus, the larger S is the less number of *eralyReshuffle* operation Ring ORAM would need and the more space it would occupy. Moreover, the S value also has effect on *evictPath* operation. A larger S leads to a larger path length and more expensive *evictPath* operations. Therefore, if we shrink the bucket size by reducing S, the number of *eralyReshuffle* increases but at the same time the cost of write path in *evictPath* decreases. To paint the full picture, we conduct a series of experiments to demonstrate the effect of S value on *eralyReshuffle* and *evictPath* and their ultimate impact on performance. Our experiments also expose the existing trade-off between space and performance.

ORAM tree is a full binary tree, and the capacity of the binary tree grows exponentially. The number of nodes of each level is equal to the number of nodes in all levels above. In a Ring ORAM tree with 24 levels, 7 last levels account for 99% of the entire capacity. Therefore, changing S for the 17 upper levels will have a negligible effect on the space demand. We change the S value for the 7 last levels. Figure 22 illustrates the results. For the baseline we followed the typical setting in [28], $Z = 12$, $Z' = 5$, and $S = 7$.

As the figure indicates, the execution time grows linearly whereas the space reduction is exponential. Hence, by shrinking the S value for the levels closer to the leaves, we can achieve a significant space reduction while incurring a low performance overhead.

In Figure 22, we reduce S value by 3 for several last levels. L-x means reducing the S

value by 3 for x last levels. As the space shrinks, the execution time increases because of the increase in memory accesses. The space saving is dampened as we move up in the tree levels toward the root because of the exponential growth of the binary tree. In $L7$ configuration, the space saving is 25% and the performance overhead is 11%. The increase in the number of *earlyReshuffle* is much more, however, it is partially covered by the fact that the *evictPath* costs less. Note that when the bucket size is reduced so is the path length, hence, the path eviction incurs less number of memory accesses.

4.3 The AB-ORAM Design

In this section, we will discuss the design of AB-ORAM.

4.3.1 Overview

In this paper, we propose AB-ORAM to reduce the space demand for Ring ORAM. We identify the source of space waste in Ring ORAM, and propose schemes to address the waste accordingly.

In the baseline ORAM, there is a block address by which the user identifies the block. Each block is mapped to a path in the tree. Each path ID is associated with a series of physical locations in the tree. In AB-ORAM, we add another level to this address redirection. To each block, we assign a logical location that identifies what path and what bucket on the path the block resides. Then, each logical location can be mapped to a physical location in the tree. We keep this mapping inside the bucket metadata. We introduce the remote allocation i.e. when the physical location of a block in the tree differs from its logical location. Note that in the baseline, logical and physical locations are always the same.

- We develop a mechanism of tracking dead blocks so that we can reuse them during the execution. The tracking process requires augmenting the metadata with extra pieces of information, and also it requires a buffer on-chip to keep the record of tracked dead blocks.

- We introduce a new allocation scheme referred to as remote allocation. Remote allocation enables the reuse of dead blocks that we constantly track over the entire ORAM tree. We implement remote allocation by adding extra (in a minimized fashion) pieces of information in the metadata.
- We decouple the slot and block information for developing the remote allocation. All slots across the tree are treated as entries in a memory pool and they can accommodate any block regardless of its path ID. We introduce a new mapping between the logical location and the physical location in the tree.
- We propose to shrink the bucket size by reducing the number of reserved dummy blocks allocated for each bucket initially. Then, to overcome the performance impact we develop a mechanism to dynamically adjust the S value for each bucket using the remote allocation.
- We shrink the bucket size of tree levels close to the leaves to further exploit the space/performance trade-off.

We now discuss each of the proposed design components and steps in detail.

4.3.2 Reclaiming Dead Blocks

To enable remote allocation, we need to keep track of the location of dead blocks in the ORAM tree so that we can later reuse them. We gather this information at the first stage of *readPath* i.e., metadata access. During the metadata access of buckets along a path, we can learn which slots are carrying dead blocks. To perform the tracking, we adopt tracking metadata, buffer, and procedures.

Tracking metadata. AB-ORAM needs to keep extra metadata in each bucket so that it can keep track of dead blocks' status and remotely allocated blocks all over the tree. Table 3 details the organization of bucket metadata in Ring ORAM and AB-ORAM. For clarity, we divide metadata fields into two categories, *block-related* and *slot-related*. The block-related metadata of a bucket contains information about the blocks that have been mapped to this bucket or in other words, the blocks whose logical location in the tree falls within this bucket. Whereas the slot-related metadata indicates information about the slot

itself, i. e. the physical location.

AB-ORAM adds four pieces of metadata to Ring ORAM: three block-related (`remote`, `remoteAddr`, and `remoteInd`) and one slot-related (`status`). The three block-related metadata are necessary for reading the blocks in *readPath*, *earlyReshuffle*, or *evictPath*. The slot-related metadata is required for tracking and reusing the slots containing dead blocks via remote allocation. Note that all metadata pieces reside in the memory like the Ring ORAM.

The `remote` flag indicates whether the corresponding block is physically present at this bucket or it resides remotely somewhere else. In other words, it indicates whether the logical and physical locations of the block are the same. In a case that they are not the same, `remoteAddr` and `remoteInd` indicate the remote bucket and slot respectively. Note that, these two fields indicate the physical location of the block in the tree. The `status` field indicates the state of the slot based on the type of the block it contains and its usage by AB-ORAM.

The `status` of all slots is initially `REFRESHED` once they are written to the ORAM tree. When a block is accessed during the *readPath*, it must be invalidated according to the Ring ORAM protocol. Thus, its `valid` flag is turned off. At this point, the `status` of the slot that contains this block becomes `DEAD` since it is carrying a dead block. A slot is marked as `ALLOCATED` when it is added to AB-ORAM's buffer.

Table 3 states the size of each metadata field in terms of ORAM parameter. Note that in this table N_{Block} and N_{Bucket} are the number of real blocks and the total number of buckets in the ORAM tree, respectively. R indicates the maximum number of slots that AB-ORAM allows remote allocation per bucket. All other parameters are identical to what is introduced in Section 4.1.3.

Tracking buffer. We want to gather the location of `DEAD` slots so that we can reuse them via remote allocation. To this end, we maintain several FIFO queues on the processor side. We refer to each of these queues as a *DeadQ*. Each *DeadQ* maintains items we call `slotInfo` that consists of two fields: {`slotAddr`, and `slotInd`}. These two identify the physical location of the slot in the tree.

As shown in Section 4.2, dead blocks at levels closer to the root tend to have a shorter

lifetime. Besides, levels closer to the root account for a small portion of the space. Thus, we skip some first levels at DRAM for gathering dead blocks, and for the rest, we dedicate a *DeadQ* to each level. Note that all of these queues are maintained on-chip.

Dead blocks have different lifetimes as we discussed in Section 4.2. Our motivational experiments have revealed that the lifetime of dead blocks varies across different tree levels by order of magnitudes. Therefore, we adopt a separate queue for each level. The reason is to maintain and organize `DEAD` slots with the same lifetime in the same queue such that a dead block with a short lifetime will not conflict with a long-lasting one. Otherwise, the flow of dead block generation/consumption in the queue will be disrupted and AB-ORAM may not be able to fully reclaim the dead block space.

Tracking procedures. We proposed a few lightweight procedures to maintain the *DeadQ*.

- `markDEAD()`: it marks the `status` of a slot as `DEAD` when its occupant block turns dead (i.e. `valid` = 0). Note that the occupant block might be either local-allocated or remote-allocated. It is invoked at the metadata access in the *readPath* operation.
- `gatherDEADs()`: it adds information of all the `DEAD` slots along a path into the corresponding *DeadQ* with the format of {`slotAddr` , `slotInd`} . Then, it marks `status` of that slot as `ALLOCATED` so that no one else will use it. It is invoked at the metadata access in the *readPath* operation.

4.3.3 Remote Allocation

We refer to each memory write as an allocation. In Ring ORAM, all allocations are in-place. Whereas in AB-ORAM, allocation may be either in-place or remote. This section discusses in detail how remote allocation works. The definition of different allocations is as follows.

- *In-place allocation*: in Ring ORAM protocol, when a block is picked to be written back to a specific bucket, it is placed into one of the Z slots of the bucket along with its metadata. We refer to this as an in-place allocation, i.e., the block's logical and physical location in the tree are the same.

- *Remote allocation*: when a block is picked to be written back to a specific bucket, its metadata is placed in that bucket. Then, a slot from the *DeadQ* is picked to accommodate the block itself. We refer to this as a remote allocation; when a write-back locates the block physically apart from its metadata. Then, the physical location of the block is kept in the metadata.

Recall that, there are two block write-back phases in the Ring ORAM protocol, one at the *earlyReshuffle*, and another one at the *evictPath* operation. At *earlyReshuffle* one or more buckets may get reshuffled. Each bucket reshuffle incurs Z memory writes. Whereas the *evictPath* incurs $L \times Z$ memory writes since it writes back an entire path. All these allocations in Ring ORAM are in-place.

Remote allocation works as follows. Suppose we want to remotely allocate block A that is originally mapped to a slot we call it *source slot*.

1. A `DEAD` slot is dequeued from the *DeadQ*. Let us call it the *target slot*.

2. Block A is then written to the *target slot*. Note that no update is needed on metadata of the *target slot*. Since *target slot* is looked up from the *DeadQ*, its `status` is already marked as `ALLOCATED` (at the time it was added to the queue) so that it will not be used by others.

3. Block-related metadata of block A is written to the metadata part of *source slot*. The `remote` flag is raised and `remoteAddr` and `remoteInd` are set to point to *target slot*. All other block-related metadata pieces are updated as they are in the Ring ORAM.

Note that, *source slot* here refers to the logical location and the *target slot* refers to the physical location of block A in the tree. Metadata always resides at the logical location and is accessible via the path ID.

Figure 23, demonstrates the remote allocation process as well as the architectural modifications that it incurs. As an example in the figure, block a on path l is accessed during the *readPath* operation. Then it becomes dead so it is added to the *DeadQ* for later reuse. Block b and c in this bucket are remotely allocated and their corresponding metadata is pointing to the remote address.

4.3.4 Dynamic Setting of S Value

The idea is to originally allocate less reserved dummy blocks (S value) for each bucket and later on try to extend the S value for each bucket dynamically by reusing the existing dead blocks in the tree. To this end, we use the remote allocation scheme that was described earlier.

AB-ORAM uses initial $Z - r$ allocation but it tries to compensate the performance degradation by dynamically increasing the S value by r for each bucket. Note that Z is the bucket size in the baseline and consists of two parameters, $Z = Z' + S$, the r reduction applies to the S parameter and keeps the Z' part intact. To minimize the performance degradation, AB-ORAM attempts to allocate each bucket with Z blocks where r blocks are provided from the dead blocks pool. In this way, AB-ORAM occupies less space while keeping the reshuffle numbers the same as the baseline. Our motivational experiments in Section 4.2 suggest that we choose $r = 2$ because the number of dead blocks of each level in the steady-state is twice the number of buckets in that level.

In Ring ORAM, for each bucket *earlyReshuffle* is activated if the bucket is touched S times. Whereas, in AB-ORAM, it depends on the current allocated S value for the bucket. Thus, AB-ORAM needs to keep a counter in the metadata of each bucket to indicate the dynamically allocated S value. We refer to it as `dynamicS` as listed in Table 3. At each bucket write (either at *earlyReshuffle* or *evictPath*), depending on the number of block allocations, `dynamicS` is determined and updated in the metadata of the bucket. In the future, `dynamicS` determines whether a bucket has to be reshuffled.

AB-ORAM aims to extend the S value at the time of allocation for every bucket. This goal can be achieved if at the time of the allocation there are enough dead blocks in the *DeadQ*. As we show in Section 4.5, there are sufficient dead blocks for the remote allocation most of the time. Figure 23 demonstrates how the dynamic S setting exploits the remote allocation. Firstly, buckets at shown levels in the figure are reduced in size. Then, the `dynamicS` is increased by 2. Pointers to the extended blocks are kept in the metadata.

4.3.5 Non-uniform Setting of S Value

As discussed in 4.2, there exists a trade-off between performance and space regarding to the S value. By reusing the dead blocks, we can compensate the performance impact. According to our experiment in 4.2, if we consider the number of existing dead blocks for each level, for the buckets we can calculate the dead block per capita. For the baseline configuration this number is 2. It means each bucket can extend its S value by 2 if we adopt remote allocation. Thus, we shrink the bucket size in original memory allocation by 2, then we extend it through remote allocation. However, if we shrink the bucket size more aggressively, by 3, 4, or 5 blocks, the number of existing dead blocks per level may not be enough to compensate S value reduction. Therefore, with remote allocation we cannot extend the S value beyond 2. However, we can exploit the space/performance trade-off and only shrink the levels close to leaves. In this way, we save up space with a small performance overhead.

4.3.6 Comparison to the State-of-the-arts

Table 4 summarizes the latest optimizations proposed on ORAM. IR-ORAM [26] improves Path ORAM performance by reducing the memory intensity of different path types. One of the key improvements in IR-ORAM is to reduce the path access overhead by shrinking the Z value for the middle levels. IR-ORAM reveals that middle levels are under-utilized. Besides, they account for a small portion of the entire capacity. Thus, shrinking the buckets of these levels improves the performance without affecting the capacity. However, it increases the probability of the stash overflow, hence, it may incur more background evictions than the baseline. IR-ORAM was proposed to optimize Path ORAM [31] while the principal can be adopted to optimize the portion of Z' entries of tree buckets in Ring ORAM.

Bucket Compaction (CB) [6] was proposed to shrink the bucket size for Ring ORAM by reducing the S value. This too reduces the path access overhead so that *evictPath* operation costs less with this scheme. Unlike IR-ORAM, CB applies the shrinking to buckets of all levels so it reduces the space demand effectively. Note that this reduction only affects the number of reserved dummy blocks (S portion of tree buckets) so the capacity of the tree

for storing real data blocks remains intact. Like IR-ORAM, it may increase the number of background evictions because real blocks may be returned to the stash instead of dummy blocks.

As discussed, we develop two schemes in this paper, i) reclaiming the dead blocks space, and ii) setting non-uniform S value across the levels. Reclaiming dead blocks allows us to reduce the space demand. We shrink the S value like CB but we compensate for the performance impact by extending the S value via remote allocation. Since remote allocation means address redirection, it may incur a slight increase in memory block accesses due to lower row buffer hit in DRAM DIMMs. Our experimental results show that this overhead is negligible. The non-uniform setting of S value also improves the space demand by shrinking the bucket size of levels close to the leaves, which helps to reduce the overhead of each *evictPath* operation. This increases the number of reshuffles for those levels. However, the reshuffle operations are off the critical path and thus exhibit a low impact on performance.

More importantly, the proposed two schemes are orthogonal to both IR-ORAM and bucket compaction. By adopting our schemes on top of IR or CB, we are able to further reduce the space demand of Ring ORAM and make it comparable to IR-ORAM. Our schemes have low performance overhead. Section 4.5 evaluates the effectiveness of our approach combining the bucket compaction.

4.4 Security Analysis

In this section, we show AB-ORAM ensures the same level of security guarantee that Ring ORAM offers. Generally, the key security in ORAM primitive is that memory accesses are indistinguishable from each other. Ring ORAM converts each user request to L block accesses, which consists of two rounds of accesses as discussed in Section 4.1.3, metadata access, and data block access. AB-ORAM issues the same number of memory accesses as Ring ORAM for both phases. Regarding the memory addresses, for the metadata access phase AB-ORAM is identical to Ring ORAM, each metadata is retrieved from its corresponding tree location address along the path. Regarding the data access phase, for remote blocks

along the path AB-ORAM uses the remote address.

The remote address by itself does not disclose any sensitive information. As a matter of fact, it adds another level of address redirection. Generally, in ORAM protocol once a block is read from the memory, its location is invalidated and no further read is allowed to that tree location until it is written again by ORAM protocol. In Path ORAM, the write-back takes place immediately after each read phase whereas in Ring ORAM the write-back is postponed to a later time for the sake of performance. Thus, Ring ORAM bears with an invalidated location in the tree for a while until one of its maintenance operations, (*earlyReshuffle* or *evictPath*) rewrites that location and refreshes it for the next read. In AB-ORAM, we propose to efficiently rewrite these invalidated locations earlier in time via remote allocation. In other words, AB-ORAM preserves the same security promise, no invalidated block is read before it is refreshed.

In Ring ORAM, the `count` field in metadata is not encrypted since the number that a bucket has been touched is already known to the public. Recall that servers adopt the direct-attached memory architecture, i.e., memory addresses are communicated off-chip in cleartext. Hence, the choice of a remote slot for allocation does not disclose any sensitive information since the location of the dead block is already public information.

Further, whether or not a remote allocation takes place provides no information to an attacker. The number of existing dead blocks is already known to the public, and AB-ORAM does not rely on any other information to decide on the remote allocation. AB-ORAM extends the bucket size in allocation only based on the availability of dead blocks.

4.5 Evaluation

To evaluate the effectiveness of AB-ORAM, we used USIMM [7], a trace-driven cycle-accurate DRAM simulator for that purpose. The simulator was widely adopted in the literature to evaluate ORAM schemes. We modeled a 4-issue OoO (out-of-order) 3.2GHz processor with 128 ROB entries. Table 5 lists the system configuration details.

To collect traces we used the Pin tool [10] SPEC CPU2017 suite [1], we used traces with

40 million memory accesses from each benchmark. For each trace, the first 38 million accesses were used to warm up the ORAM tree, and the last two million were fed to USIMM for DRAM access simulation. While not reported, we ran longer traces and the performance results remained stable. Table 6 lists the benchmarks in the experiments, which consist of both integer and floating-point benchmarks. The table indicates L2 misses per kilo instruction (MPKI) for each benchmark.

We modeled a Ring ORAM tree with 24 levels, $Z = 12$, and $Z' = 5$, $S = 7$, i.e., the tree occupies $2^{(24-1)}\times12\times64$B=12GB memory space. Following the prior work [31, 41, 42, 25, 26, 6], the protected user data occupies around 50% of all Z' entries in buckets, that is, $2^{(24-1)}\times5\times50\%\times64$B=2.5GB. We adopted a tree cache that saves top 10 levels on-chip [26].

The schemes we evaluated are as follows. Note that they are built on top of a Ring ORAM baseline.

- `CB`: It implements bucket compaction on Ring ORAM [6], with $Y = 4$, $Z = 8$, $Z' = 5$, and $S = 3$.
- `CB + IR`: It implements IR-ORAM utilization optimization [26] on top of `CB`. It sets $Z' = 3$ for levels in range [L10, L15] and $Z' = 4$ for [L16, L18].
- `CB + DS`: It implements dead block reclamation for dynamic S setting on top of `CB`. It sets the bucket size to $Z = 6$ ($Z' = 5$, and $S = 1$) for levels in range [L18, L23], then extend the S value by 2 dynamically via remote allocation.
- `CB + NS`: It implements non-uniform S setting on top of `CB`. It sets the bucket size to $Z = 6$ ($Z' = 5$, and $S = 2$) for levels in range [L21, L23].
- `CB + AB`: It combines our two schemes, `DS` and `NS` on top of `CB`. It sets $Z = 6$ ($Z' = 5$, and $S = 1$) for levels in range [L18, L20] and $Z = 5$ ($Z' = 5$, and $S = 0$) for levels in range [L21, L23].

4.5.1 Performance and Space Analysis

Figure 24 illustrates the normalized execution time of each scheme compared to the baseline. In addition, it shows the amount of space occupied by each scheme compared to the baseline. `CB` improves the execution time of Ring ORAM by 10% and it occupies 67%

of the Ring ORAM space. `CB + IR` exhibits less performance improvement of 4%. The reason is both schemes increase the need for background eviction which may dampen the performance improvement. Note that the improvement of utilization optimization in IR-ORAM is much more significant when adopted on top of Path ORAM. In Ring ORAM bucket size is larger because of reserved dummy accesses so the percentage of path length reduction is lower. Besides, in Ring ORAM path length does not affect the online access overhead, it only lowers the cost of the *evictPath* operation. Since `IR` only shrinks the Z' value for the middle levels, it has a negligible effect of less than 1% on the space demand. `CB + DS` lowers the space demand to 50% of the baseline. The execution time of this scheme is only 2% more than `CB` alone. With `CB + NS` the space demand is reduced to 59% of the baseline and it exhibits the same execution time of `CB` alone. With this scheme, the number of reshuffles is increased but the cost of path eviction is reduced. This scheme sets S value non-uniformly across the levels, also our experiments reveal that it is beneficial if we adopt a non-uniform Y value when combining our scheme with `CB`. Thus, in `CB + NS` for the levels that S is reduced by one, Y is also reduced by one. This prevents the increase of green block usage that may cause extra background eviction. `CB + AB` has the lowest space demand of 43%. Its execution time is 3% more than the `CB` alone while it has 24% more space reduction than `CB` alone.

4.5.2 Background Eviction Overhead

`CB` enables background eviction in Ring ORAM. It is necessary to prevent stash overflow. Unlike Path ORAM, path eviction does not occur after every online access. Instead, it occurs after every five accesses. To prevent information leakage, CB has to preserve this pattern. Thus, when background eviction is needed, it generates dummy *readPath* operations until it is time for an *evictPath*. Figure 25 shows the breakdown of normalized online accesses compared to the baseline Ring ORAM. In general, dummy accesses constitute small portion of online accesses. `CB` has dummy ratio of 0.5%. `CB + DS` has a higher ratio of 1.8%. `CB + NS` has only 0.3% dummy accesses. For `CB + DS` the dummy ratio is higher than `CB` because the initial bucket allocation in this scheme is $S = 1$ which is more aggressive than $S = 3$

in CB. Thus, before DS has the chance to extend the S value, more green blocks may be brought to the stash and consequently increase the need for background eviction. CB + AB that combines our two schemes, consists of 1.6% dummy accesses on average. It is slightly lower than CB + DS because with NS being adopted the Y value is also set non-uniformly as discussed earlier.

4.5.3 Bucket Reshuffle Impact

Figure 26 compares the reshuffle number of different schemes across the levels. CB has the same number of reshuffles as the baseline. CB + DS has a slightly higher number of reshuffles, however, still it keeps it close to the baseline. CB + NS almost doubles the number of reshuffles for the levels in the range [L21, L23] where the S value is reduced. CB + AB has the highest number of reshuffles for the levels in the range [L21, L23] since it has the most aggressive setting of $S = 0$ for those levels.

4.5.4 DS Sensitivity Analysis

We ran a sensitivity analysis to evaluate our design across the choice of the level range for our dynamic S scheme. Figure 27 shows the performance result of different configurations as well as their normalized space demand. Note that DS-L18 in the figure is the same as CB + DS in Figure 24. Conceptually, the top levels are less desirable for remote allocation because their contribution to the space demand is low. For instance, the top 17 levels account for less than 1% of the space while their reshuffle number contributes the same to the performance.

4.5.5 Remote Allocation Effectiveness

Figure 28 indicates the ratio of extended bucket size compared to total number of bucket allocations. As shown in Section 4.2, there are abundant dead blocks available at each level, on average 2 dead blocks per bucket. Therefore, DS is able to extend almost all of the bucket allocations by 2. In contrast, when NS is also enabled, there is less number of dead blocks available. Thus, CB + AB has lower extending ratio of 74%. However, still the majority of

bucket allocations can benefit from the S value extension.

4.5.6 Storage Overhead

The remote allocation incurs two types of storage overhead, one is the tracking buffer to maintain information of dead block locations in the tree for remote allocation, and another one is extra metadata at the buckets. The former occupies on-chip storage overhead whereas the latter incurs extra space in the external memory. This section discusses each in detail.

On-chip Overhead. As discussed before, for the tracking buffer we dedicate a *DeadQ* for each level that we gather its `DEAD` slots. We skip the first 8 levels in DRAM for gathering the candidate slots for remote allocation. We perform the gathering for level 18 to level 23. A *DeadQ* is dedicated to each level. We set the size of *DeadQ*s empirically to 1000 entries. Note that a *DeadQ* only requires to store tree locations for dead blocks. In total *DeadQ*s need 21 KB on-chip space.

Memory Overhead. With the configuration of Ring ORAM baseline, the bucket metadata takes 33 B that fits into a block (i.e. 64 B (a cache line)). To avoid incurring any performance penalty during the metadata access phase we keep the extra added metadata by AB-ORAM less than a block size by tuning the R parameter in Table 3. R is the maximum number of remote allocations allowed per bucket. Our results have shown $R = 6$ is sufficient for effectively extending the bucket size via remote allocation. Our scheme incurs 28 B of extra metadata. Thus, in total, bucket metadata will fit in one 64 B block.

4.6 Related Work

Ring ORAM is the most bandwidth-efficient ORAM protocol. Recent studies have been proposed to improve the performance of Ring ORAM. Cao *et. al.* proposed String ORAM to reduce Ring ORAM overhead via spatial and temporal optimization schemes [6]. Their spatial scheme is the bucket compaction. The temporal scheme introduces a more efficient scheduling approach for handling memory commands to accelerate Ring ORAM. Che *et. al.*

proposed a channel imbalance-aware scheduler [8] to minimize the channel imbalance for read requests in Ring ORAM. Devadas *et. al.* proposed Onion ORAM [11] to construct a constant bandwidth blowup scheme. It leverages poly-logarithmic server computation to avoid the logarithmic lower bound on ORAM bandwidth blowup. Chen *et. al.* implemented Onion Ring ORAM [9] to outperform logarithmic-bandwidth ORAM such as Ring ORAM. Hoang *et. al.* developed a distributed ORAM scheme [18] to achieve a client storage efficiency in addition to efficient client-server bandwidth.

Many optimizations also have been proposed over the Path ORAM protocol. Maas *et al.* proposed to buffer a few top levels on-chip to reduce number of memory accesses[24]. Yu *et al.* proposed PrORAM to construct super-blocks that enforces neighbouring blocks to be mapped to the same path, it then improves Path ORAM performance by enabling prefetching [41]. Zhang *et al.* proposed to reduce accessing overhead by eliminating overlapped portion of consecutive path accesses [43]. Wang *et al.* proposed to improve performance by mitigating the interference of Path ORAM applications on co-running processes on the same server [36]. Zhang *et al.* proposed to exploit the dummy blocks of the same path to save duplicate copies of the blocks further in the path so that the processor can resume execution early [42]. Wang *et al.* proposed to adopt Path ORAM for emerging BOB (buffer-on-board) memory architecture [37]. Nagarajan *et al.* proposed to construct an extra smaller ORAM tree that acts as an intermediate cache between LLC and main ORAM tree such that majority accesses can be served by the smaller tree, which reduces the average length Path ORAM access. [25].

Fletcher *et al.* proposed Freecursive to store a unified ORAM tree that combines both the position map and the data so that ORAM accesses are not distinguishable [13]. To achieve the highest level of security, Fletcher *et al.* proposed to defend timing channel attacks by issuing path accesses at fixed rate and pattern where dummy accesses are issued if no real user request awaits [14].

Recent studies have proposed to adopt Path ORAM for protecting data storage servers [30, 22]. Memory access patterns can also be effectively protected by adopting hardware enhancements [2, 3] . Hardware-assisted security schemes are also developed to mitigate the impact of data authentication on performance [34].

4.7 Conclusion

In this paper, we propose AB-ORAM to address the space inefficiency of Ring ORAM. AB-ORAM identifies two sources of space waste, accumulation of dead blocks after online accesses that hold useless data till the next refresh, and large buckets have low performance benefit for tree levels close to the leaves. AB-ORAM reduces the space demand by reclaiming the dead blocks space via remote allocation. It furthers the space reduction by setting a non-uniform bucket size. AB-ORAM shrinks the buckets close to the leaves. We built our schemes on top of the latest optimization of Ring ORAM and effectively lower its space demand. AB-ORAM makes the space demand of Ring ORAM comparable to the most space-efficient ORAM implementations while preserving the Ring ORAM performance benefit. AB-ORAM achieves an average of 36% space reduction over the state-of-the-art while introducing very low performance overhead.

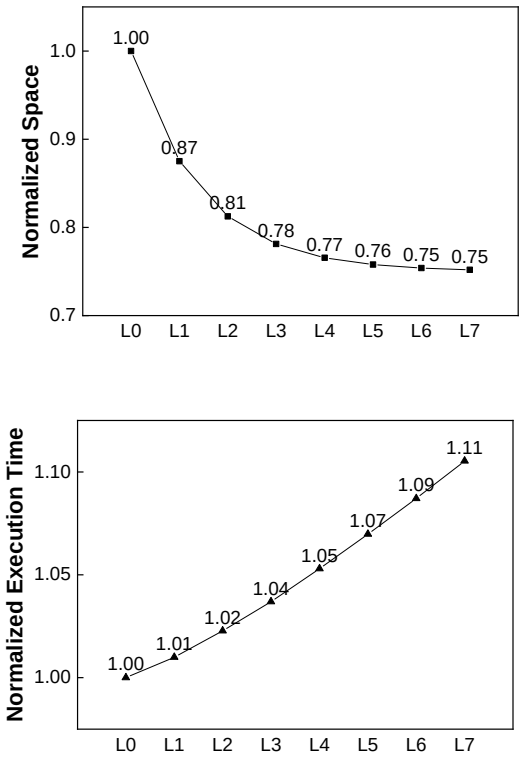


Figure 22: Space/Performance trade-off, reducing S by 3 across the bottom levels. Space demand across different path length normalized to baseline (top), execution time normalized to the baseline (bottom).

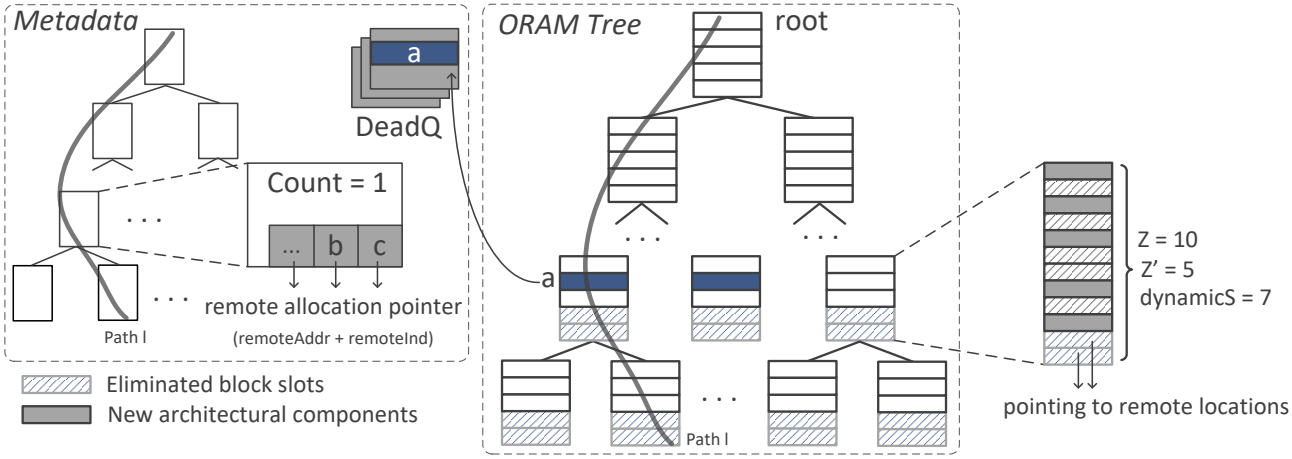


Figure 23: An overview of remote allocation in AB-ORAM.

	Ring ORAM [28]	IR-ORAM [26]	Bucket Compaction [6]	This work	
				Dead block reclaim	Non-uniform S value
Space demand	-	improved	improved	improved	improved
Online access	-	-	-	slight more	-
Bucket reshuffle	-	-	-	slight more	more
Path eviction	-	-	improved	slight more	improved
Background eviction	-	more	more	-	-

Table 4: Summary of the state-of-the-art ORAM implementations.

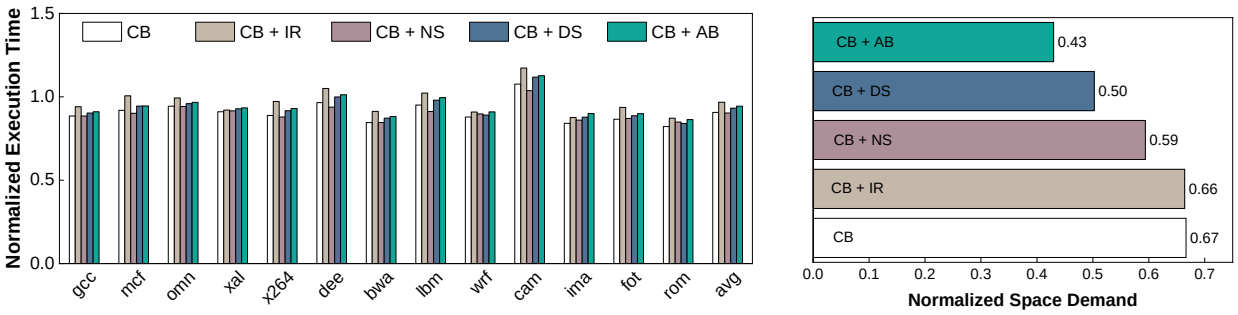


Figure 24: Performance and space comparison of different schemes.

Processor Configuration	
Processor Fetch Width/ ROB Size	4 / 128
Memory Channels	4
DRAM Clk Frequency	800 MHz
L1 D-cache	2-way 256KB
L2 cache (LLC)	8-way 2MB
ORAM Configuration	
Protected user data	2.5 GB
ORAM tree levels	24
Bucket size/Block size	4 / 64B
Stash entries	300
Dedicated tree top cache	256KB (4K entries)
On-chip PLB / PosMap	64KB / 512KB

Table 5: System configuration.

Integer Benchmark	read MPKI	write MPKI	Float Benchmark	read MPKI	write MPKI
gcc	0.1	0.3	bwa	0.0	20.7
mcf	19.5	0.1	lbm	0	45.3
xz	24.9	29.6	wrf	0.2	1.3
xal	0.05	0.1	cam	0.01	8.8
x264	1.3	1.2	ima	0.3	2.9
dee	0.0	5.7	fot	0.02	1.5
			rom	0.02	23.0

Table 6: Evaluated benchmarks.

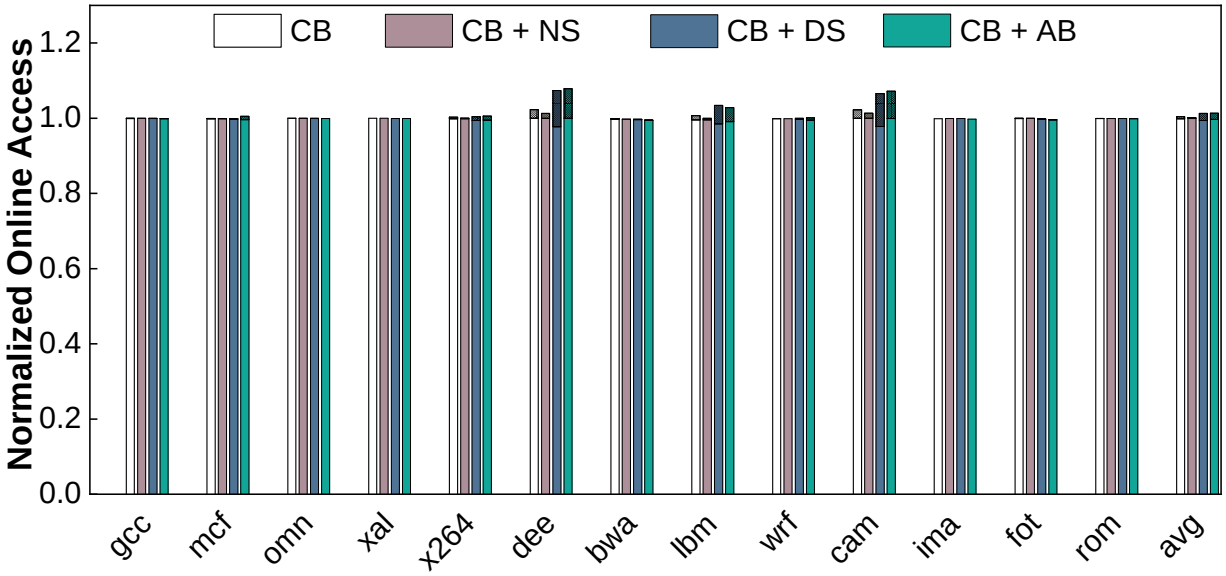


Figure 25: Breakdown of online accesses in different schemes.

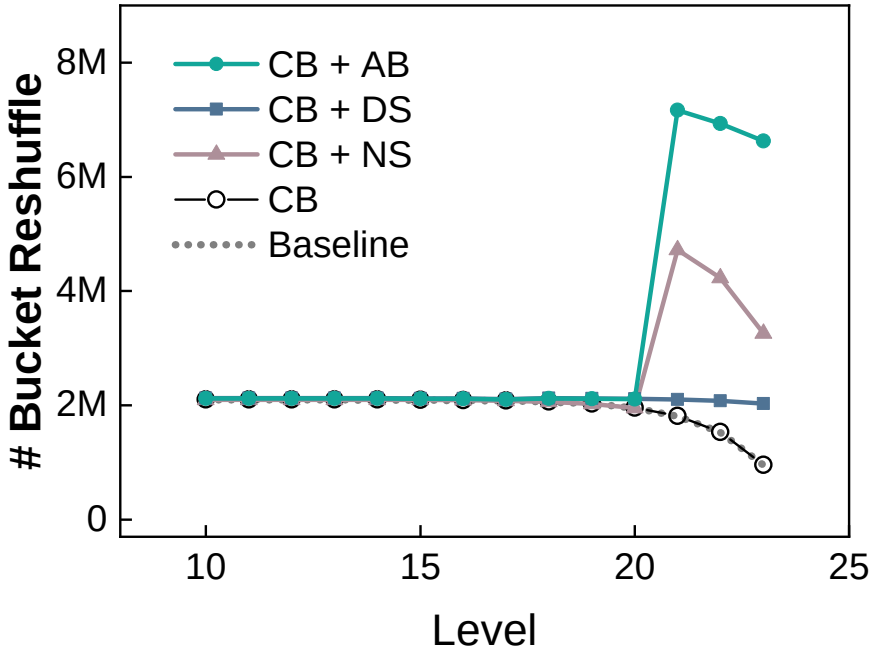
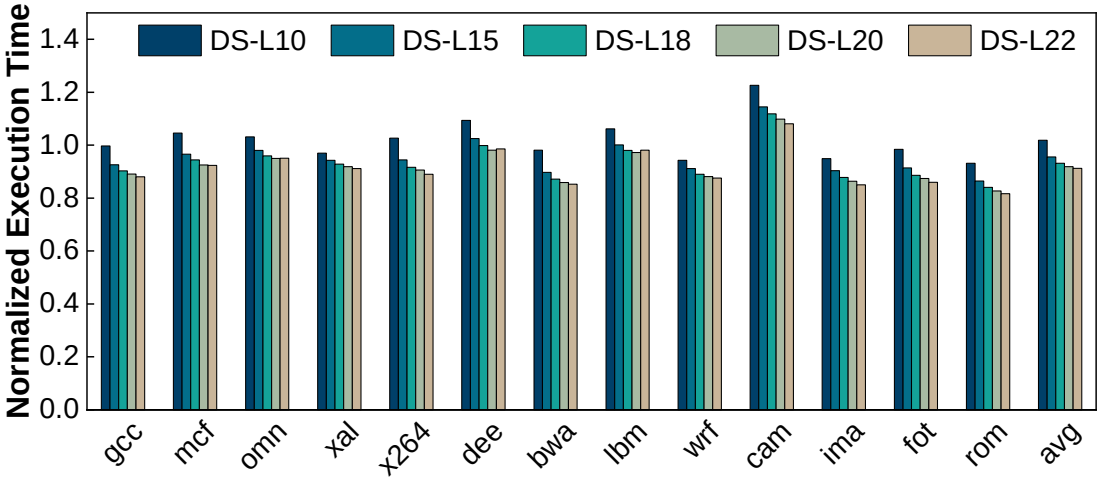


Figure 26: Comparing number of reshuffles across the levels.



Normalized Space Demand				
DS-L10	DS-L15	DS-L18	DS-L20	DS-L22
50.00%	50.03%	50.26%	51.04%	54.16%

Figure 27: Sensitivity analysis of DS to the number of levels.

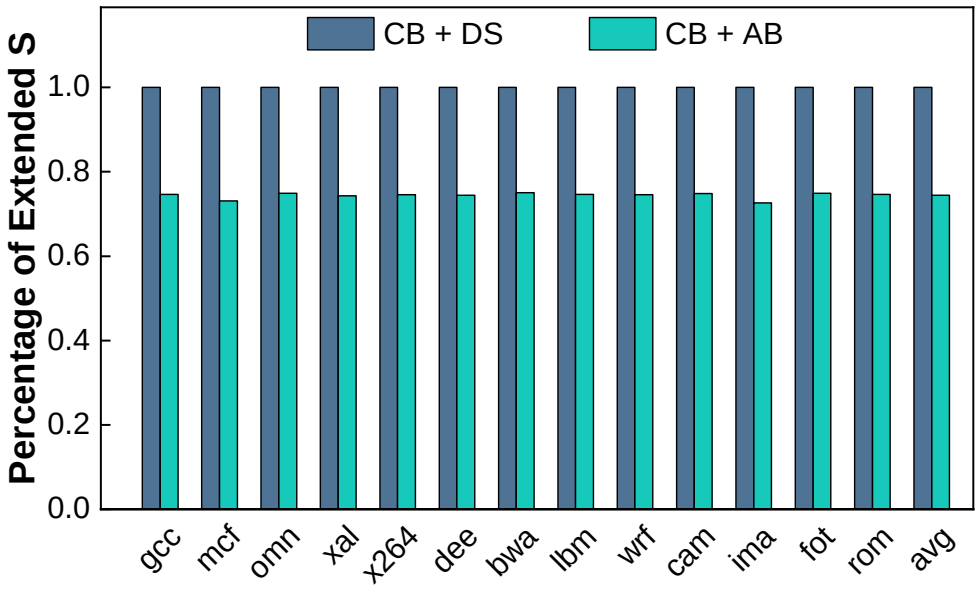


Figure 28: AB-ORAM capability for extending S value.

5.0 Future Work

5.1 Key Observations

In Section, we discussed the distribution of hits across the different tree regions in Path ORAM. We realized the top region of the ORAM tree gets even more hits in Ring ORAM. Figure 29 demonstrates this phenomenon. the X-axis indicates the different settings of top tree region boundaries. Lx means up to level x is considered the top region. The boundary between the middle and bottom is the same for all the configurations and it equals 20. It means levels 21 to 23 are considered bottom and the rest (levels x+1 to 20) are considered middle levels.

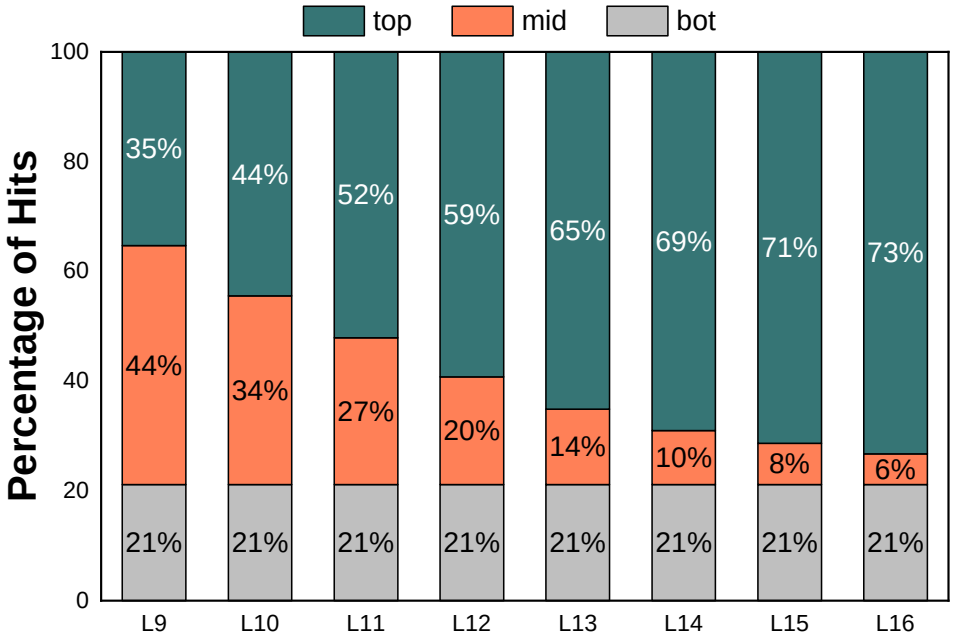


Figure 29: Hit distribution .for different top region boundaries.

To investigate the reason we conduct an utilization analysis over the Ring ORAM implementation similar to the experiment in Section . Figure 30 illustrates the overall average of bucket utilization across the levels in Ring ORAM. Note that the result is showing the aver-

age of 13 benchmarks. We observe that the trend is almost the same for all the benchmarks. For this experiment we use a 400 million trace for each benchmark.

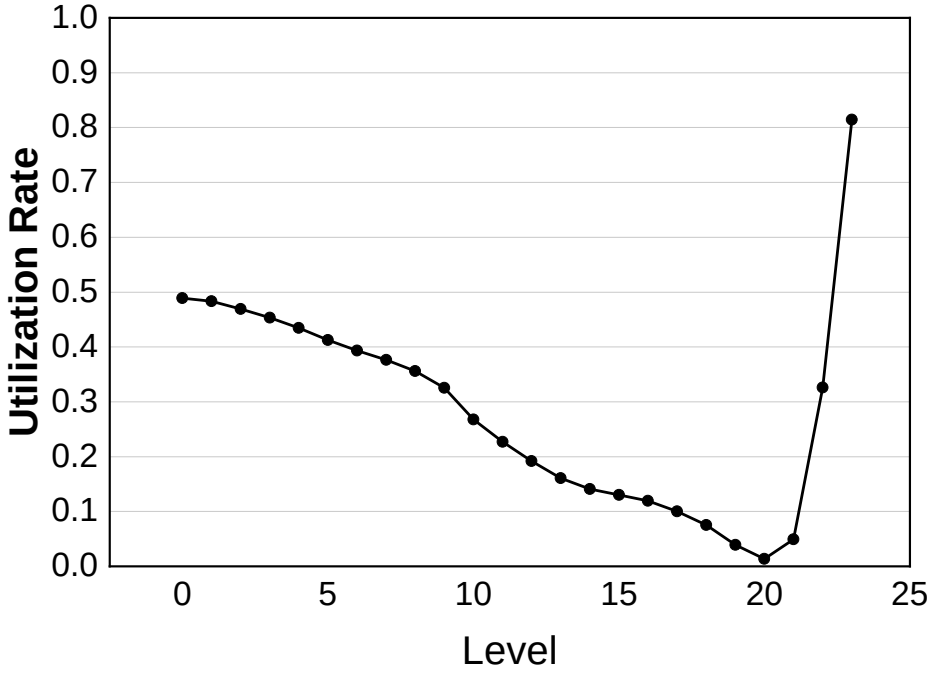


Figure 30: Hit distribution for different top region boundaries.

Comparing this figure to , we realize that top levels tend to have a higher average utilization in Ring ORAM. This is expected since the Ring ORAM adopts a lazy write-back operation. As mentioned before, in Ring ORAM, after every 5 online access, one path eviction takes place. In the meantime, the accessed real blocks remain in the stash. When a path eviction happens, it is more likely for the blocks in the stash to be moved to the top region of the tree as discussed in Section 5.1. Therefore, given the delayed path eviction in Ring ORAM, it is natural that we observe higher utilization in the top region and consequently high hit. We observe more hits because application programs have temporal locality. Thus, when recently accessed blocks remain longer in the top region, they have a higher chance to be reused while they are still on-chip. The average utilization can vary over the benchmarks but the trend is the same. In all 13 benchmarks that we experimented with, we observed the top region in Ring ORAM gets more hits than Path ORAM counterpart. Figure 30 shows the average of 13 benchmarks.

In another experiment, we try to increase the utilization of treetop deliberately. To

this end, we alternatively skip the levels in treetop during the path eviction. Figure shows the utilization rate across the levels for this experiment. As the figure indicates, treetop utilization is increased. Moreover, it confirms that the reason for Ring ORAM seeing a higher hit in the treetop is the delayed path eviction.

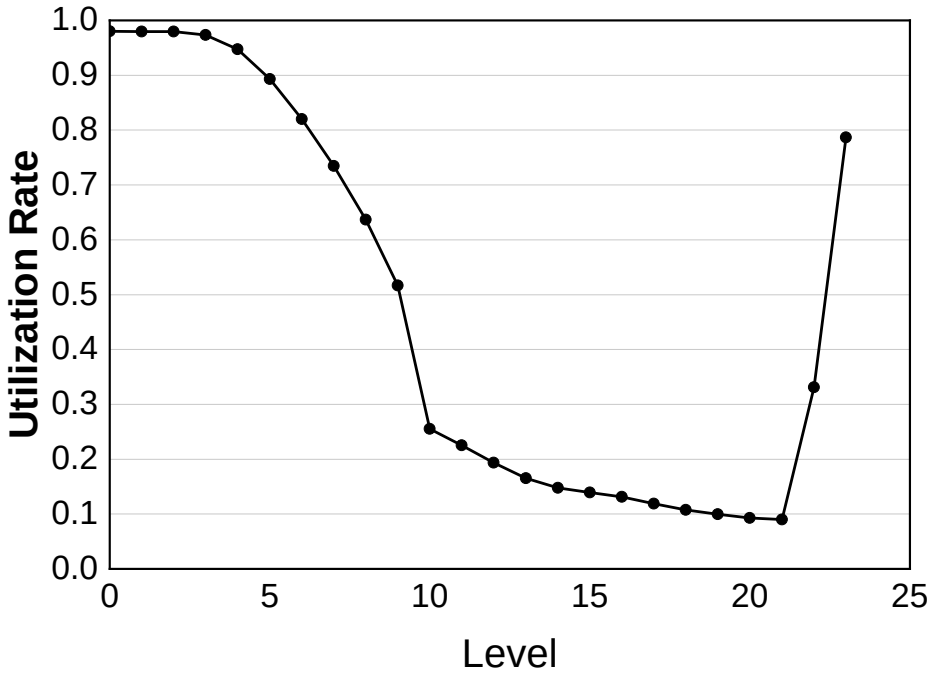


Figure 31: Hit distribution for different top region boundaries.

5.2 Proposed Schemes

5.2.1 Short Path/ Long Path Eviction

As Figure 30 indicates, middle levels are highly underutilized. We propose to exploit this phenomenon in order to reduce the overhead of path eviction. The proposal is to alternatively skip the bottom levels in path eviction. In a way that, every other path eviction we access the half path instead of the full path. One implication is that middle levels will experience a higher utilization and it is feasible because the middle levels are already highly underutilized. It might incur a higher stash occupancy if during the half path eviction the middle levels

have no empty slot.

Figure 32 shows the result of skipping the bottom levels (below 19). As expected, the utilization for middle levels has slightly increased. Our results indicate the number of memory accesses decreases by 13% when this scheme is applied.

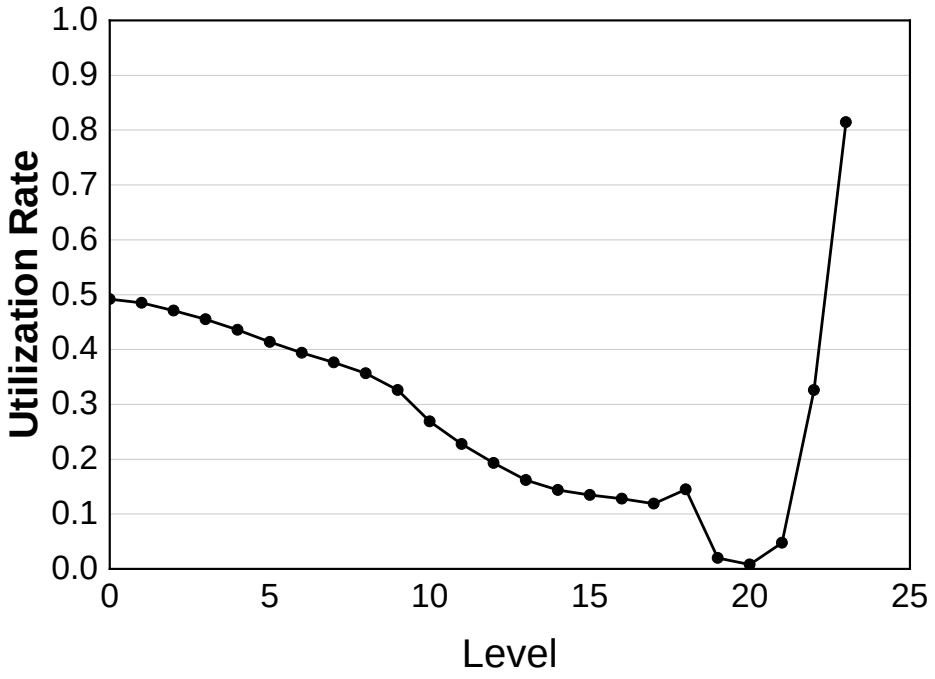


Figure 32: Hit distribution for different top region boundaries.

5.2.2 Treetop Independent Eviction

To overcome the stash overflow, we propose a new scheme other than background eviction. Recall that in background eviction of Ring ORAM, dummy online accesses are generated until it is time for a path eviction. Background eviction is more expensive in Ring ORAM because of all the compulsory dummy accesses. We propose an on-chip path eviction scheme to free up the stash space. In the scheme, we consider the top tree cache as an independent tree and we perform path eviction on that until the stash occupancy is dropped below a threshold. Since the top tree resides on-chip, there is no limitation on the number and pattern of path eviction. Thus, we can issue as many path evictions in a row as needed.

Note that none of the proposed schemes compromise the security of Ring ORAM. As

long as path accesses have the same pattern, they leak no information to the attacker. Our first scheme is to issue half path and full path eviction to reduce the overhead. This incurs no security issue because the attacker always sees that every other path eviction is half path access and this pattern remains the same during the execution and is the same for all application programs. Our second idea does not raise any security concerns either. Since the entire top region is cached on-chip, the pattern of issuing path eviction discloses no information to the attacker as it incurs no off-chip access.

Bibliography

[1] *SPEC CPU 2017 Benchmark Suite*, 2017.

[2] Shaizeen Aga and Satish Narayanasamy. Invisimem: Smart memory defenses for memory bus side channel. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

[3] Amro Awad, Yipeng Wang, Deborah Shands, and Yan Solihin. Obfusmem: A low-overhead access obfuscation for trusted memories. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture*, 2017.

[4] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas. Cacti 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization*, 2017.

[5] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[6] Dingyuan Cao, Mingzhe Zhang, Hang Lu, Xiaochun Ye, Dongrui Fan, Yuezhi Che, and Rujia Wang. Streamline ring oram accesses through spatial and temporal optimization. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[7] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. H. Pugsley, A. N. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti. Usimm : the utah simulated memory module. 2012.

[8] Yuezhi Che, Yuan Hong, and Rujia Wang. Imbalance-aware scheduler for fast and secure ring oram data retrieval. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, 2019.

[9] Hao Chen, Ilaria Chillotti, and Ling Ren. Onion ring oram: Efficient constant bandwidth oblivious ram from (leveled) tfhe. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[10] Intel Corp. *Pin - A Dynamic Binary Instrumentation Tool*, 2012.

[11] S. Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, E. Shi, and Daniel Wichs. Onion oram: A constant bandwidth blowup oblivious ram. In *TCC*, 2016.

[12] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. Advanced encryption standard (aes), 2001.

[13] C. W. Fletcher, L. Ren, A. Kwon, M. van Dijk, and S. Devadas. Freecursive oram: [nearly] free recursion and integrity verification for position-based oblivious ram. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[14] C. W. Fletchery, 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 *2014 IEEE 20th International Symposium on High Performance Computer Architecture*, 2014.

[15] Blaise Gassend, E Suh, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. Caches and merkle trees for efficient memory authentication. In *Proceedings of 9th International Symposium on High Performance Computer Architecture*, 2003.

[16] Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, 1987.

[17] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43, 1996.

[18] Thang Hoang, Ceyhun D. Ozkaptan, Attila A. Yavuz, Jorge Guajardo, and Tam Nguyen. S^3oram: A computation-efficient and constant client bandwidth blowup oram with shamir secret sharing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[19] Intel. *Intel Software Guard Extensions*, 2014.

[20] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. 2007.

[21] Hsien-Hsin S. Lee, Gray S. Tyson, and Matthew K. Farrens. Eager writeback- a technique for improving bandwidth utilization. In *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2000.

[22] Liang Liu, Rujia Wang, Youtao Zhang, and Jun Yang. H-oram: A cacheable oram interface for efficient 1/o accesses. In *2019 56th ACM/IEEE Design Automation Conference*, 2019.

[23] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[24] 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 *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013.

[25] Chandrasekhar Nagarajan, Ali Shafiee, Rajeev Balasubramonian, and Mohit Tiwari. ρ: Relaxed hierarchical oram. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[26] Mehrnoosh Raoufi, Youtao Zhang, and Jun Yang. Ir-oram: Path access type based memory intensity reduction for path-oram. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[27] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[28] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Ring ORAM: closing the gap between small and large client storage oblivious RAM. *IACR Cryptol. ePrint Arch.*, 2014.

[29] Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. Design space exploration and optimization of path oblivious ram in secure processors. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[30] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. Zerotrace : Oblivious memory primitives from intel SGX. In *25th Annual Network and Distributed System Security Symposium*, 2018.

[31] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013.

[32] Jeffrey Stuecheli, Dimitris Kaseridis, David Daly, Hillery C. Hunter, and Lizy Kurian John. The virtual write queue: coordinating dram and last-level cache policies. 2010.

[33] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Aegis: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th Annual International Conference on Supercomputing*, 2003.

[34] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. VAULT: reducing paging overheads in SGX with efficient integrity verification structures. In Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar, editors, *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[35] David Lie Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[36] Rujia Wang, Youtao Zhang, and Jun Yang. Cooperative path-oram for effective memory bandwidth sharing in server settings. In *2017 IEEE International Symposium on High Performance Computer Architecture*, 2017.

[37] Rujia Wang, Youtao Zhang, and Jun Yang. D-oram: Path-oram delegation for low execution interference on cloud servers with untrusted memory. In *2018 IEEE International Symposium on High Performance Computer Architecture*, 2018.

[38] Zhe Wang, Samira M. Khan, and Daniel A. Jiménez. Improving writeback efficiency with decoupled last-write prediction. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[39] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, 2015.

[40] Jun Yang, Youtao Zhang, and Lan Gao. Fast secure processor for inhibiting software piracy and tampering. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[41] Xiangyao Yu, Syed Kamran Haider, Ling Ren, Christopher Fletcher, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Proram: Dynamic prefetcher for oblivious ram. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015.

[42] X. Zhang, G. Sun, P. Xie, C. Zhang, Y. Liu, L. Wei, Q. Xu, and C. J. Xue. Shadow block: Accelerating oram accesses with data duplication. In *51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018.

[43] Xian Zhang, Guangyu Sun, Chao Zhang, Weiqi Zhang, Yun Liang, Tao Wang, Yiran Chen, and Jia Di. Fork path: Improving efficiency of oram by removing redundant memory accesses. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture*, 2015.

[44] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. Hide: An infrastructure for efficiently protecting information leakage on the address bus. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.