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

2024

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF COMPUTER SCIENCE

This dissertation was presented

by

Mehrnoosh Raoufi

It was defended on

March 29, 2024

and approved by

Dr. Youtao Zhang, Department of Computer Science

Dr. Xulong Tang, Department of Computer Science

Dr. Stephen Lee, Department of Computer Science

Dr. Jun Yang, Department of Electrical and Computer Engineering

Improving Performance and Space Efficiency of Secure Memory with ORAM

Mehrnoosh Raoufi, PhD

University of Pittsburgh, 2024

In modern computing, safeguarding user privacy has become a major challenge. Data content can be protected via encryption. However, encryption cannot hide the location of the data being accessed and thus may leak the access pattern to the adversary. Studies have shown that the access pattern can leak sensitive information about the user such as medical records, user identity, and financial information. In contemporary computer systems, the processor chip integrates a memory controller on-chip and transmits memory addresses and device commands in cleartext on memory buses. Therefore, even if the data is encrypted, one can snoop on the bus and untrusted off-chip memory and infer sensitive information via observing the memory address trace of a user program. ORAM (Oblivious RAM) is an expensive cryptographic technique that obfuscates access patterns. When ORAM is implemented in the processor, it converts each off-chip memory request from a user program into tens to hundreds of memory accesses. While ORAM provably hides the access pattern, thereby bringing privacy protection, from the system perspective, it incurs significant overhead. ORAM requires continuously shuffling user data in the memory. Not only does it slow down the execution of user programs, but it also imposes substantial memory space demand. This dissertation aims to minimize ORAM overhead from different aspects by introducing various architectural techniques. The goal is to make ORAM more desirable for wide adoption, thereby enabling modern computing systems with efficient, privacy-preserving secure memory. The dissertation analyzes the most popular ORAM implementations and their latest optimizations in the literature to expose ORAM inefficiencies in terms of performance, and space demand. Then, it strives to alleviate these inefficiencies. It reduces the memory intensity of ORAM by decreasing the number of memory accesses needed per user request, thereby reducing bandwidth overhead and improving ORAM performance. It also reduces ORAM memory space demand by reclaiming invalidated memory blocks in ORAM and saving further DRAM space through the introduction of a hybrid-memory ORAM design.

Table of Contents

Preface xii
1.0 Introduction 1
1.1 The Case for Protecting Access Patterns 1
1.2 ORAM Overhead and Challenges 3
1.3 Contribution Overview 6
2.0 Background 11
2.1 Memory Security 11
2.1.1 Memory Basics 11
2.1.2 Security and Privacy 12
2.2 ORAM Basics 14
2.2.1 ORAM History 14
2.2.2 Related Work 14
2.2.3 Threat Model 16
2.3 Path ORAM 17
2.3.1 Operation Basics 17
2.3.2 Key Enhancements 19
2.3.3 Ring ORAM Optimization 20
3.0 IR-ORAM: Path Access Type Based Memory Intensity Reduction for Path-ORAM 24
3.1 Intensity Reduction Overview 24
3.2 Motivation 26
3.2.1 The Types of Path Accesses 26
3.2.2 The Utilization of Tree Nodes 28
3.2.3 The Block Migration Behavior 30
3.3 The IR-ORAM Design 33
3.3.1 An Overview 33

3.3.2 IR-Alloc: a Utilization-aware Node Size Allocator for Reducing Intensity of PT$_p$/PT$_d$ /PT$_m$ paths 33
3.3.3 IR-Stash: a Double Indexed Stash Implementation for Reducing Intensity of PT$_p$ Paths 36
3.3.4 IR-DWB: Converting Dummy Path Accesses for Reducing Intensity of PT$_m$ paths 39
3.3.5 Security and Correctness Analysis 42
3.4 Experiment Methodology 44
3.5 Experimental Results 45
3.5.1 Performance Comparison 46
3.5.2 IR-Alloc Overflow 48
3.5.3 PosMap Reduction 50
3.5.4 Dummy Path Accesses 50
3.5.5 Scalability Analysis 52
3.5.6 Overheads 53
3.5.7 Conclusion 53
4.0 AB-ORAM: Constructing Adjustable Buckets for Space Reduction in Ring ORAM 55
4.1 Space Reduction Overview 55
4.2 Ring ORAM Space Demand 57
4.2.1 Ring ORAM with Bucket Compaction 57
4.2.2 Broad Impact of Space Reduction 58
4.3 Motivation 58
4.3.1 Studying Dead Blocks 59
4.3.2 Studying Space/Performance Trade-off 61
4.4 The AB-ORAM Design 62
4.4.1 Overview 62
4.4.2 Remote Allocation 63
4.4.2.1 One Extra Level of Address Mapping 63
4.4.2.2 Tracking Dead Blocks 65

4.4.2.3 Remote Allocation 68
4.4.3 Altering the S Value for Space Savings 68
4.4.3.1 Extending the S Value with Remote Allocation 68
4.4.3.2 The Non-uniform S Value 70
4.4.4 Comparison to the State-of-the-arts 71
4.5 Security and Correctness 72
4.5.1 Remote Allocation is Secure 72
4.5.2 Altering the S Value is Secure 73
4.5.3 Empirical Security Analysis 74
4.5.4 Correctness 75
4.6 Experimental Methodology 76
4.7 Experimental Results 78
4.7.1 Main Results of Space and Performance 78
4.7.2 Bucket Reshuffle Impact 79
4.7.3 DR Sensitivity Analysis 81
4.7.4 Dead Block Lifetime Analysis 81
4.7.5 NS Design Exploration 82
4.7.6 Remote Allocation Effectiveness 82
4.7.7 Generalizability Over Different Applications 84
4.7.8 Storage Overhead 84
4.7.9 Conclusion 85
5.0 EP-ORAM: Efficient NVM-Friendly Path Eviction for Ring ORAM in Hybrid Memory 86
5.1 DRAM Saving Overview 86
5.2 EP-ORAM 88
5.2.1 Trade-offs of Adopting ORAM in DRAM/NVM Memory 88
5.2.2 Trade-offs among Ring ORAM Operations 89
5.2.3 Under-utilized Middle Levels in Ring ORAM Tree 90
5.2.4 The EP-ORAM Design 91
5.2.5 Security Analysis 93

5.3 Evaluation 95
5.3.1 DRAM Space, NVM Traffic and Performance Analysis 96
5.3.2 EP-ORAM Design Exploration 98
5.3.3 Utilization Analysis 98
5.3.4 Conclusion 101
6.0 Conclusions 102
6.1 Summary 102
6.2 Future Work 103
6.2.1 ORAM for Recommendation Systems 103
6.2.2 Merkle Tree for Position Map Access Reduction 105
Bibliography 110

List of Tables

1 System configuration for IR-ORAM evaluation. 44
2 Evaluated benchmarks from SPEC and PARSEC suite. 45
3 Organization of bucket metadata in Ring ORAM and AB-ORAM. 67
4 Summary of the state-of-the-art ORAM implementations. 71
5 System configuration for AB-ORAM evaluation. 77
6 Evaluated benchmarks of SPEC suite. 77
7 System configuration for EP-ORAM evaluation. 95

List of Figures

1 ORAM incurs a large performance degradation. 4
2 Each user request is translated to multiple path accesses. 5
3 ORAM's space demand significantly burdens the system. 6
4 Contribution of each level to memory accesses and to the size of the ORAM tree. 8
5 Overview of performance and space efficiency improvements. 10
6 The threat model adopted in the dissertation. 17
7 An overview of Path ORAM ($L = 3$, $Z = 4$). 19
8 Ring ORAM tree organization ($L = 3$, $Z' = 3$, $S = 4$, and $Z = 7$). 21
9 Ring ORAM tree organization and operations with $L = 3$. 23
10 The distribution of different path accesses. 27
11 The space utilization at different tree levels. 28
12 The space utilization behavior per benchmark; gcc (left), lbm (middle), and a random trace (right). 30
13 The migration behavior after block a leaves the stash. 31
14 Nodes at top levels have high reuse possibility. 32
15 The design of IR-Alloc scheme. 34
16 Exploit IR-Stash to effectively cache large tree top. 36
17 The design of IR-DWB scheme. 40
18 The performance comparison of different schemes. 46
19 The performance comparison with LLC-D as baseline. 47
20 Design exploration of IR-Alloc scheme. 48
21 Level utilization with IR-Alloc. 49
22 Comparing PosMap accesses with Baseline. 51
23 Access type distribution in IR-DWB. 51
24 The scalability analysis of IR-Alloc. 52
25 Dead blocks over time for different benchmarks. 60

26 Dead blocks across the levels. 60
28 AB-ORAM adds one more level of address mapping in the insecure domain. . . 64
29 An overview of remote allocation in AB-ORAM. 65
30 Empirical study on AB-ORAM security implication. 75
31 Space saving and performance overhead comparison of different schemes. 79
32 Bandwidth impact of AB-ORAM. 80
33 Comparing number of bucket reshuffles across the levels. 80
34 Sensitivity analysis of DR to the number of levels. 81
35 Dead blocks lifetime across ORAM tree levels. 82
36 Design exploration of NS. 83
37 AB-ORAM capability for extending the S value. 83
38 Generalizability analysis of AB-ORAM. 84
39 Slowdown of Ring ORAM in hybrid memory compared to DRAM (Hybrid-NVMx indicates saving x bottom levels in NVM). 88
40 Bucket utilization across Ring ORAM tree levels (Utilization=1.0 means saving Z' blocks in one bucket). 90
41 An overview of the EP-ORAM design in hybrid memory system. 92
42 DRAM space demand of different schemes. 96
43 NVM writes reduction of EP-ORAM compared to Hybrid-NVM2. 97
44 Performance comparison of EP-ORAM and Hybrid-NVM2. 97
45 NVM writes reduction with different configurations of EP-ORAM compared to Hybrid-NVM3. 99
46 Performance improvement with different configurations of EP-ORAM compared to Hybrid-NVM3. 99
47 Comparing EP-ORAM and Hybrid-NVM4 slowdown. 100
48 Bucket utilization of EP-ORAM and the baseline across levels. 100
49 Timeline of path access in a large embedding setup with a 4-level PosMap table. 106
50 The PosMap block layout with the Merkle tree PosMap scheme. 108
51 The total memory access reduction achieved with the Merkle PosMap scheme. . 109

Preface

PhD, as much as being a Doctor of Philosophy, means doctor of perseverance to me. It is about having faith in exploring the unknown for the slightest potential of an idea and having the courage to walk away once it has been proven ineffective. I have learned that the latter is as important as the former. One cannot achieve scientific expertise unless one remains impartial to facts and findings. Oscillating between thinking my research could change the world and feeling that my research does not matter, it has been quite a roller coaster, but it was a wonderful journey that was worth the ride. It is not about changing the world, but rather about contributing your part, no matter how insignificant it may seem. These are the insights I will carry into my future career now that this journey has reached its end.

I would like to express my gratitude to all the people who helped me during the PhD program. Sincere thanks to the academic community that made this experience possible. I would like to thank my advisor, Professor Youtao Zhang. He shaped my scientific thinking. I will cherish the countless insightful and in-depth discussions we have had over the years that profoundly enriched my problem-solving paradigm. I am grateful to him for all the accomplishments I was able to achieve during my PhD. It would not have been possible without his guidance. My thanks also go to Professor Xulong Tang for all his help, guidance, and encouragement. I would also like to express my gratitude to the rest of my dissertation committee: Professor Stephen Lee and Professor Jun Yang for their valuable input and constructive feedback.

I would like to thank the community of the University of Pittsburgh. I will remember my time there fondly. My heartfelt appreciation goes to this university for caring about students' mental and physical well-being. I would also like to thank all my friends. They made it possible for me to survive far away from home.

To my family, I dedicate this dissertation. For their love is my cornerstone. I would like to thank them for their infinite and unwavering support, and for always believing in me. I would not have reached this point without them. Though they live far away, they are always in my heart and my drive to keep moving forward.

1.0 Introduction

1.1 The Case for Protecting Access Patterns

Outsourcing of data and computation is an indispensable aspect of this digital era. In today's context, users entrust cloud service providers with their data more than ever before. This reliance extends not only to individuals but also to businesses such as insurance corporations and banks, which utilize remote servers for both computation and data storage. Consequently, safeguarding user data becomes paramount. Regarding data protection, one might instinctively turn to encryption as the ultimate solution. While encryption, coupled with authentication schemes, can indeed shield the content of user data, it alone proves insufficient in thwarting adversaries from inferring sensitive information. The issue lies in access pattern leakage.

Many research studies have shed light on the privacy risks associated with access pattern leakage, even when the data content is completely encrypted and protected. There are numerous works revealing the threat of access pattern attacks on searchable encrypted data servers. Researchers have demonstrated that by exploiting the access pattern, an untrusted server holding encrypted user data can leak sensitive information. This includes users' queries or the secret attributes of every record in the database, such as salary, age, or location. In some cases, it may also reveal the correlation between a secret attribute and other attributes in the database, such as name, gender, or occupation. Ultimately, an adversary could infer confidential information regarding users' personal preferences, financial status, or health conditions, posing a significant threat to user privacy [36, 8, 38, 27, 39, 28, 41].

The pattern of a program's memory accesses can reveal much about its behavior or the encrypted data it processes. For instance, the memory access pattern of a program can be used to reconstruct the program control flow graph, which can lead to the identification of the program being executed. Leakage of the program control flow information can potentially lead to the compromising of a secret key. This is because conditional branches make a comparison between two values and then decide which path to take. Thus, knowledge of the program control flow can reveal information regarding the values being compared in a conditional branch instruction [86].

One recent study has shown that exploiting access patterns of sparse features in deep learning-based recommendation systems can lead to private data leakage. The study characterizes various attacks that can be carried out on these sparse features, leading to the extraction of users' private information or tracking users over time. The study shows that even when data is fully encrypted, an attacker can still learn information about which entries of the sparse features are non-zero through the embedding table access pattern. This information leakage can pose a significant threat to user privacy. The study has shown that it is possible to identify a user, extract sensitive attributes of a user, or re-identify a user, by only looking at the embedding table access pattern, even when the data within the embedding table is fully encrypted [31].

As discussed above, the threat of access pattern leakage affects data storage servers, program execution in trusted processors while communicating with untrusted memory, and embedding tables in deep learning-based recommendation systems. In all cases, access pattern leakage poses a significant threat to user privacy. Therefore, to safeguard user privacy, one must pay attention to access pattern protection. Recent studies have shown that to fully protect the access pattern, ORAM (Oblivious RAM) must be adopted. Although ORAM offers bullet-proof access pattern obfuscation and is particularly appealing from a security point of view, it comes with significant overhead. This, in turn, makes ORAM less desirable for widespread adoption from a system perspective.

The goal of this dissertation is to improve how ORAM utilizes system resources. While many of the techniques introduced in this dissertation are applicable to other configurations as well, the focus of this dissertation and its evaluation methodology is on the secure processor and untrusted main memory configuration. The next section explains the overhead and challenges associated with implementing ORAM in modern computing environments. The subsequent section enumerates the contributions of this dissertation towards mitigating those overheads.

1.2 ORAM Overhead and Challenges

Oblivious RAM (ORAM) is a cryptographic primitive which obfuscates the access patterns [25, 26]. ORAM aims to safeguard a user program's memory access pattern when interacting with untrusted external memory on a remote server. Despite data encryption, transmitting addresses in plain text exposes vulnerabilities, enabling adversaries to track accessed locations and potentially leak sensitive user information. To fortify complete user privacy, protecting the memory access pattern is crucial. Nevertheless, ORAM has a huge overhead. Not only does it slow down the user program execution, but it also imposes a large memory space and bandwidth overhead.

At a very high level, ORAM keeps data items randomly shuffled in memory. When the user wants to access one data item, ORAM translates it into many accesses to different memory locations. After the access, the data item is remapped to another location; the blocks that have already been read are reshuffled, re-encrypted, and written back to memory. Therefore, ORAM converts each memory request from the user's program into tens to hundreds of memory accesses. And since it needs to constantly shuffle data blocks around, it requires allocating some extra space in memory.

Path ORAM [66] 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 [25, 26] 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 $O(\log N)$ since the data blocks are organized in a full binary tree.

A variety of Path ORAM optimizations have been proposed over the years, all attempting to mitigate the high overhead associated with Path ORAM. Nevertheless, Path ORAM remains a highly memory-intensive primitive that incurs significant performance overhead. Path ORAM, equipped with its key enhancements (which will be discussed in Section 2.3.2), as illustrated in Figure 1, incurs an average slowdown of 8× compared to non-secure execution.

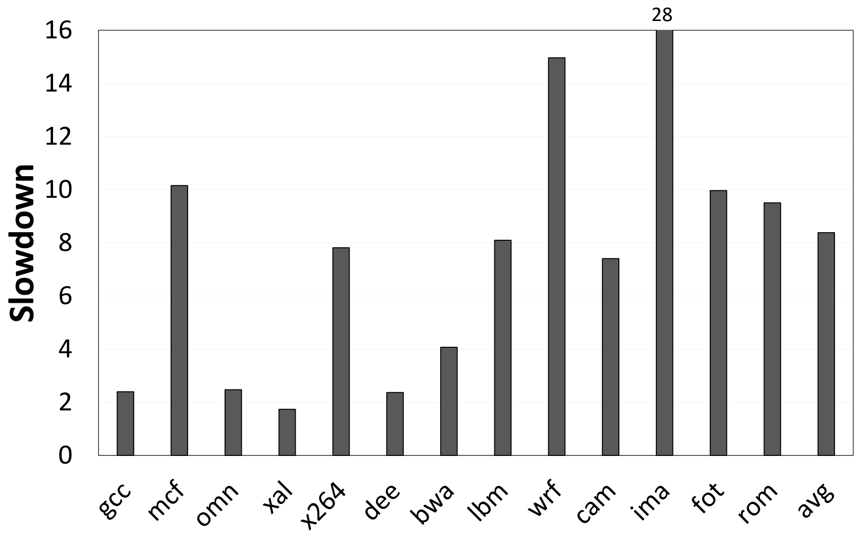


Figure 1: ORAM incurs a large performance degradation.

There are two main reasons behind Path ORAM performance degradation: 1) each user request translates into one or multiple path accesses, and 2) each path access is expensive, involving hundreds of memory accesses. Figure 2 illustrates the average number of path accesses required for each benchmark in our experiment settings. As shown in the figure, for some benchmarks, this number can be as high as 3.6 path accesses per user request.

In Path ORAM, each access consists of two phases: the read path phase and the write path phase. The read phase involves reading all the memory blocks along a path in the ORAM tree, including both real and dummy blocks. Similarly, the write phase entails writing a number of memory blocks equal to the length of the entire path to the ORAM tree. In this protocol, half of the bandwidth is on the critical path of execution; to service the user's request, an entire path is read from the ORAM tree. It is only after the read path phase is completed that the block of interest can be returned to the user who requested it. The block transfer of this portion of the access, i.e., the read path phase, is on the critical

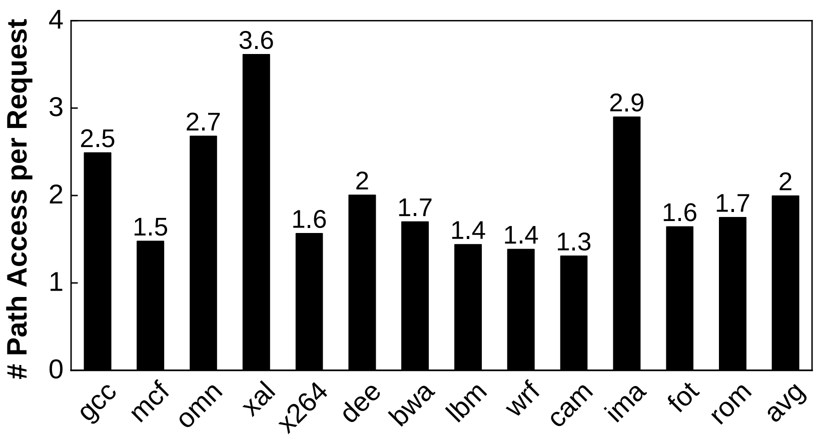


Figure 2: Each user request is translated to multiple path accesses.

path of execution.

Ren *et al.* proposed a different approach to achieve a breakthrough in speeding up the Path ORAM protocol at the expense of extra memory space [58]. They proposed Ring ORAM, which speeds up Path ORAM by 2.7× while increasing the space demand. Next, the discussion will delve into the philosophy behind the performance vs. space demand trade-off exploited in devising their design on top of Path ORAM.

Ring ORAM optimizes Path ORAM protocol by distinguishing between the online and offline bandwidth phases of ORAM tree access and deferring some memory accesses from the online phase to the offline phase. The online bandwidth refers to the portion of the block transferred until the user's request is served. In other words, it is the amount of blocks transferred until the access is complete from the user's point of view. In Ring ORAM design, the number of memory accesses in the online phase is reduced and accessed memory blocks remain invalidated until they are refreshed by one of Ring ORAM's write-back operations later on during the offline phase. From the user's perspective, this approach ensures that access is completed more quickly, allowing the user program to resume earlier. This results in a speedup of the overall system performance. However, it incurs significant space overhead. Figure 3 demonstrates that in a typical setting, as discussed in [58], user data occupies only 20.8% of the ORAM tree space, with the remainder dedicated to dummy space, which constitutes the majority of the memory space.

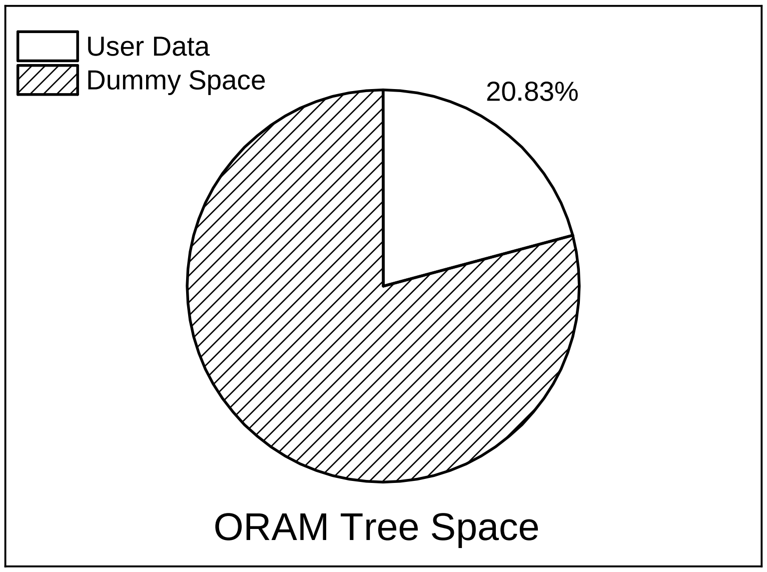


Figure 3: ORAM's space demand significantly burdens the system.

1.3 Contribution Overview

As discussed in the previous section, performance and space overhead of ORAM are two challenges that can affect each other. The optimization of the ORAM protocol, which achieves considerable performance improvements, wastes memory space resources. While the speedup may be appealing, the large space demand hinders widespread adoption in secure processor configurations, given that memory is a particularly precious resource in these systems. To progress ORAM toward practicality, I formulate the following questions:

(i) Can we speed up ORAM's performance without excess space demand?

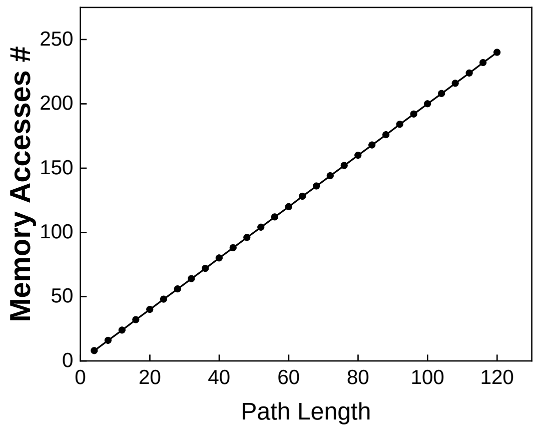
(ii) Can we minimize ORAM's space demand without sacrificing performance improvements?

In pursuit of an answer to question (i), an exploration of the Path ORAM protocol was undertaken to pinpoint the root cause of its memory intensity. Analysis was conducted on all types of path accesses within this protocol. Subsequently, efforts were made to alleviate the costs linked with each type of path access, aiming to diminish the memory intensity of ORAM and thereby enhance its performance. The introduction of IR-ORAM is the outcome of this study, which is a scheme consisting of three techniques for reducing memory intensity.

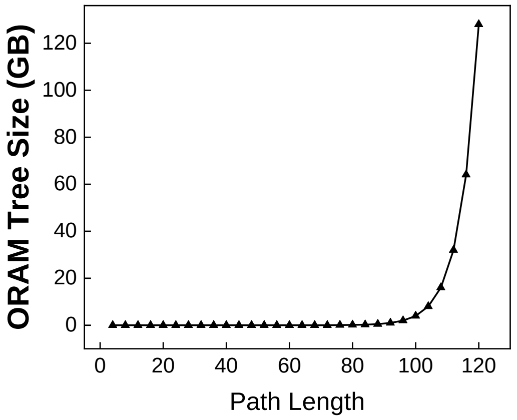
In pursuit of an answer to question (ii), an investigation into Ring ORAM optimization was conducted to identify the source of space wastage. Subsequently, AB-ORAM was introduced, employing two techniques to reclaim wasted space during execution, thus enhancing the space efficiency of ORAM. Additionally, exploration was undertaken into a hybrid DRAM/NVM memory configuration to further conserve precious DRAM space while implementing Ring ORAM. This exploration led to the introduction of EP-ORAM, an NVM-friendly ORAM design.

The mission of this dissertation is to exploit opportunities for improving the performance and space efficiency of ORAM. Although there are numerous key observations that have led to the contributions of this dissertation, which I will elaborate on in the corresponding chapters, there is one key insight worth mentioning here as it guides schemes for both objectives. The insight is that the number of memory accesses grows linearly with the path length. In contrast, the ORAM tree size grows exponentially with respect to the path length, as shown in Figure 4a and Figure 4b, respectively. This implies that a reduction in the bucket size at the upper levels of the tree can result in considerable performance improvement while having a negligible effect on the ORAM tree capacity. Conversely, reducing the bucket size of a few bottom levels can result in a significant reduction in space while having a negligible impact on performance. This key insight—the contrast in the contribution of each level to the number of memory accesses versus space—is one of the main insights that has illuminated the opportunities leading to the contributions of this dissertation in terms of improving the performance and space efficiency of ORAM. The following will provide an overview of the contributions made in this dissertation.

Performance. As discussed in the previous section, Path ORAM performance overhead stems from two main issues: each user request is translated into multiple path accesses, and



(a) Number of memory accesses per path access grows linearly w.r.t. the path length.



(b) The ORAM tree size grows exponentially w.r.t. the path length.

Figure 4: Contribution of each level to memory accesses and to the size of the ORAM tree.

each path access, in turn, consists of hundreds of memory accesses, making it very expensive and degrading the overall performance of program execution. Therefore, the goal of this dissertation is to analyze all types of path accesses in Path ORAM and to reduce both the intensity of each access and the overall number of path accesses. This is achieved by introducing a set of three techniques:

- *IR-Alloc*, which reduces the number of data blocks accessed for each path by introducing a utilization-aware node size allocation strategy.
- *IR-Stash*, which reduces the number of PosMap accesses by introducing a dual stash design that mitigates PosMap accesses for data blocks present in the treetop cache.
- *IR-DWB*, which reduces the number of dummy path accesses by converting them into useful write-back operations of dirty entries in the last level cache (LLC).

This dissertation presents these intensity reduction techniques under the title IR-ORAM.

Space. The focus is on Ring ORAM optimization to address its significant space overhead, nearly five times that of user data. The main cause of Ring ORAM's space inefficiency is the presence of reserved dummy blocks (S blocks per bucket). Despite this, these blocks offer a bandwidth/performance benefit. Extra dummy space enables Ring ORAM to tolerate more online accesses before requiring offline actions, such as reshuffling and refreshing invalidated blocks (also known as dead blocks).

The space demand issue is addressed with two schemes in this dissertation: 1) Analysis of the source of space inefficiency in Ring ORAM and proposing techniques to overcome those inefficiencies; 2) Proposing an NVM-friendly Ring ORAM implementation on DRAM/NVM hybrid memory to achieve further DRAM space saving.

In the first scheme, the progression of dead blocks over time and the correlation of the S parameter with performance over the ORAM tree levels were studied. The outcome of these analyses led to the identification of two inefficient uses of memory space in Ring ORAM: (i) accessed blocks holding useless data until the next reshuffle operation; and (ii) large buckets providing a diminishing performance benefit for tree levels close to the leaves. Two techniques were then proposed to exploit the optimization opportunities, respectively. Specifically, accessed blocks are reclaimed early by allocating them to buckets that need a reshuffle, and the bucket size for tree levels close to the leaves is shrunk for a better space/performance trade-off. These techniques are presented under the title AB-ORAM.

In the second scheme, two key observations were made: (i) for a tree-based Ring ORAM memory organization, saving bottom levels in NVM can dramatically reduce the DRAM memory requirement; (ii) the trade-offs among Ring ORAM operations expose design opportunities without security compromise. Therefore, the scheme involves saving the bottom levels of the ORAM tree in NVM and shortening the path of the EvictPath operation, which not only mitigates the number of NVM writes but also speeds up execution.

Figure 5 illustrates the trade-off between space efficiency and performance, showing how improvements in one aspect tend to degrade the other among existing ORAM implementations (i.e., Path ORAM and Ring ORAM). Space efficiency is defined as the inverse of the amount of DRAM space required per user data, while performance is defined as the inverse of execution time. Ring ORAM demonstrates the highest performance, but low space efficiency compared to Path ORAM. The figure also positions works from this dissertation within this space. IR-ORAM matches Path ORAM's space efficiency while improving performance. AB-ORAM closely aligns with Path and Ring ORAM in both space efficiency and performance, while EP-ORAM has slightly lower performance than Ring ORAM for better space efficiency, even surpassing Path ORAM. The works presented in this dissertation aim to bridge the gap between achieving the best performance and the most space efficient ORAM implementations.

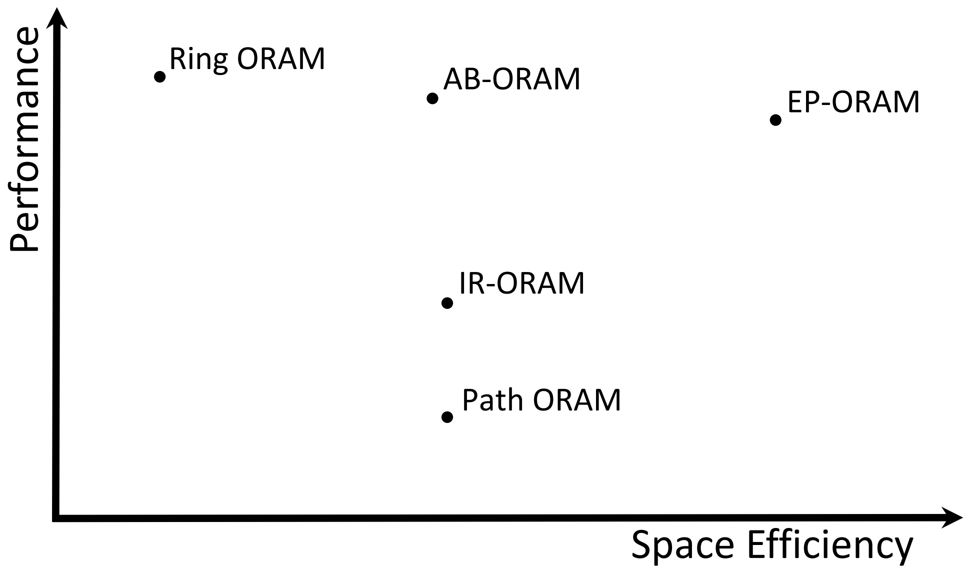


Figure 5: Overview of performance and space efficiency improvements.

The rest of the dissertation is organized as follows. Chapter 2 provides the background of ORAM and details various implementations, specifically discussing the basics of Path ORAM and Ring ORAM operations.

Chapter 3 introduces IR-ORAM. It discusses various path access types in Path ORAM and analyzes the memory intensity of each type. Following this analysis, a set of three techniques is described to address the intensity of each path type.

Chapter 4 introduces AB-ORAM. The chapter delves into analytical experiments uncovering space waste in Ring ORAM, which is followed by a discussion on how to utilize these observations to mitigate space demand in Ring ORAM.

Chapter 5 introduces EP-ORAM. The chapter discusses how a DRAM/NVM hybrid memory design can be exploited to save DRAM space while adopting the ORAM protocol.

Chapter 6 concludes the dissertation by summarizing the contributions and providing direction for future work.

2.0 Background

2.1 Memory Security

2.1.1 Memory Basics

Main memory is a key component in modern computing systems. It is part of the memory hierarchy working with the CPU (Central Processing Unit). The memory hierarchy is the organization of different types of memory based on their speed, cost, and capacity. The goal is to provide the CPU with fast access to the data while balancing cost efficiency and physical space constraints. The higher levels are faster, smaller in capacity, and more expensive, while the lower levels are slower, larger, and cheaper. The memory hierarchy typically consists of the following levels:

- **Registers.** At the top of this hierarchy, there are CPU registers located inside the CPU. They are small storage units that contain the data the CPU is currently processing. Modern processors typically have 32 to 64 registers [33]. Registers are the highest-speed memory units in the hierarchy.
- **Cache.** In the middle of the hierarchy, positioned between the CPU and main memory, lies the cache. Cache memory significantly speeds up data access for the CPU by storing frequently accessed data and instructions. It operates on the principle of temporal locality, where recently accessed data is likely to be accessed again in the near future. Caches are organized into multiple levels—L1, L2, and L3—where L1 is the closest to the CPU and offers the fastest access times but has the smallest capacity. L2 and L3 caches provide progressively larger storage capacities and slower access times, but are still significantly faster than accessing the main memory. Cache sizes range from tens of kilobytes to a few megabytes for the last level cache [33].
- **Main memory.** Located further down the hierarchy, main memory acts as a larger pool of data storage that the CPU can access directly. It has a significantly larger capacity than the previous levels, with capacities up to tens of gigabytes [33]. Main memory holds

the operating system, applications, and files that are in active use.

The hierarchy of volatile data ends at the main memory; beyond the main memory, we have disk storage such as Hard Disk Drives (HDDs) and Solid-State Drives (SSDs). Unlike main memory, these devices retain the data even when the power is turned off. In the subsequent discussion, main memory technology is examined in greater detail owing to its significance within this dissertation.

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 [37].

2.1.2 Security and Privacy

In this section, the foundation of secure processing is discussed, with an illustration of how access pattern protection fits into the picture.

Outsourcing computation and data has become more prevalent than ever today. With outsourcing arises the concern for user privacy and security. Cloud servers need to address the security and privacy implications of outsourcing for their users. That is why secure processors like Intel's Software Guard Extensions (SGX) have emerged. Intel SGX is a set of instructions that are built into some Intel processors in order to provide users with a trusted execution environment. The main goal is to provide a mechanism to execute the critical parts of a program in a secure environment so that it is protected from the rest of the system, including the operating system and any other application with root privilege. This isolation is meant to protect the code from disclosure and tampering.

Intel SGX ensures secure processing by utilizing enclaves, which are protected areas of memory for code execution, and employing attestation to verify authenticity. It further enhances data security through sealing, encrypting data with keys specific to the enclave for secure storage, thereby ensuring that only authorized enclaves can access it. However, it cannot guard against access pattern attacks; although the data is encrypted, the pattern in which the data is accessed may reveal information about the data itself or the executing program.

Commodity DRAM employs a direct-attached memory architecture, where addresses are communicated in plaintext on the memory bus. This setup allows for a straightforward scheme of address mapping, resolving physical addresses into DRAM indices such as channel ID, rank ID, bank ID, row ID, and column ID. However, this transparency means that anyone snooping the memory bus can learn which memory locations the processor accesses, potentially leaking information about user programs to external attackers. Recent research has highlighted a solution to this vulnerability: the use of a cryptographic primitive known as Oblivious RAM (ORAM). ORAM protects the address access pattern by obfuscating it, reshuffling, and re-encrypting the data blocks after each memory access, thus safeguarding against such leaks.

2.2 ORAM Basics

2.2.1 ORAM History

This section describes the origins of ORAM (Oblivious RAM) and its evolution over time. The concept of ORAM was first formulated by Goldreich in 1987 [25], with the goal of protecting against software IP theft. Although cryptographic techniques can secure the contents of memory, they do not conceal the access patterns of program execution. A naive solution for obfuscating the program's control flow involves scanning the entire memory for each access, which is highly inefficient in terms of both performance and bandwidth. Goldreich proposed the Square Root ORAM to achieve better performance. In this design, to protect N data blocks, $\sqrt{N}$ dummy blocks are stored alongside the real data blocks in the memory, with all blocks encrypted and permuted. To further improve the bandwidth efficiency of Square Root ORAM, Goldreich and Ostrovsky introduced the hierarchical ORAM in 1996 [26]. This approach organizes server storage into a pyramid of permuted arrays that grow exponentially in size, reducing the amortized bandwidth cost to $O(\log^3 N)$. A significant advancement came in 2011 when Shi *et al.* proposed the first tree-based ORAM protocol [64], marking a turning point in ORAM's evolution. The most notable among tree-based ORAM protocols is Path ORAM, which reduces the ORAM overhead to $O(\log N)$, established in 2013. Since its publication, Path ORAM has set a benchmark for protecting access patterns in secure processor configurations. Section 2.3 will provide a detailed discussion on this protocol.

2.2.2 Related Work

By adopting secure memory architectures, memory access patterns on the buses can be effectively protected with hardware enhancements [2, 4]. In addition to defending user privacy, hardware-assisted security enhancements are developed to mitigate the performance impact of data authentication [70].

Path ORAM was also enhanced for its adoption for cloud services. Recent studies have proposed to adopt ORAM for protecting data storage servers [63, 44]. Stefanov *et* *al.* proposed ObliviStore, a high-performance, distributed ORAM-based cloud data store [65]. Chakraborti *et al.* introduced rORAM to minimize the number of random physical disk seeks when accessing ranges of sequentially logical blocks in an untrusted storage [9]. ConcurORAM was proposed to improve overall throughput with a parallel, multi-client oblivious ORAM [10]. Williams *et al.* introduced PrivateFS that is an oblivious file system based on a parallel ORAM [75]. Radix Path [3] was proposed on top of Path ORAM to reduce the space demand. In this scheme, the root node is expanded to a 1000-entry node to buffer nodes of other levels and all other buckets are reduced in size. It requires a very expensive background eviction to avoid path overflow. Sahin *et al.* proposed TaoStore, an oblivious data storage that processes client requests concurrently and asynchronously in a non-blocking fashion [61]. Hoang *et al.* developed a distributed ORAM scheme [34] to achieve client storage efficiency in addition to efficient client-server bandwidth. Maiyya *et al.* propoesed QuORAM [47] that presents a Quorum-Replicated Fault-Tolerant ORAM Datastore, enhancing reliability in oblivious RAM systems. By employing quorum replication, it ensures fault tolerance and maintains data integrity even in the presence of node failures.

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 [21]. 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 [22]. Nagarajan *et al.* proposed to construct an extra smaller ORAM tree to reduce the average length of Path ORAM access [49]. demand of Ring ORAM [7]. Their spatial scheme is the bucket compaction. Yu *et al.* proposed PrORAM to improve Path ORAM performance by constructing superblocks for prefetching [81]. Zhang *et al.* proposed to eliminate accessing overlapped portion of consecutive paths [83]. Wang *et al.* proposed to mitigate the interference of Path ORAM on co-running processes on the same server [72]. Cao *et al.* proposed String ORAM to reduce Ring ORAM overhead via spatial and temporal optimization schemes to reduce execution time and space The temporal scheme introduces a more efficient scheduling approach for handling memory commands to accelerate Ring ORAM. Maas *et al.* proposed constructing a treetop cache to reduce the number of memory accesses and hence improve overall performance. Treetop caching is effective because the binary tree grows exponentially, as a result, many levels can be cached in a very small on-chip space[46]. Zhang *et al.* proposed to save duplicate copies in the dummy blocks to receive the data block early [82]. Wang *et al.* proposed to adopt Path ORAM for emerging BOB (buffer-on-board) memory architecture [73]. Che *et al.* proposed a channel imbalance-aware scheduler [12] to minimize the channel imbalance for read requests in Ring ORAM. Devadas *et al.* proposed Onion ORAM [18] 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 [14] to outperform logarithmic-bandwidth ORAM such as Ring ORAM. Liu *et al.* proposed PS-ORAM [43] to offer efficient crash consistency for ORAM protocols adopted in NVM. Rajat *et al.* proposed PageORAM [52] that is a DRAM page-aware ORAM strategy that optimizes access patterns for increased security and reduced overhead, leveraging the granularity of DRAM page accesses to enhance system performance.

2.2.3 Threat Model

This dissertation adopts the same threat model as other existing ORAM studies [66, 21, 59, 72, 49, 82, 81]. The focus is 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. Figure 6 depicts the threat model adopted in this dissertation. 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.

Data secrecy and data integrity are assumed to be protected with hardware-assisted security enhancements. [71, 35, 68, 24, 79]. 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 [35]. A Merkle tree is built on the user data to prevent unauthorized changes [24]. Existing studies showed that the performance impacts of these enhancements are effectively mitigated with the trustworthy crypto hardware integrated in the processor.

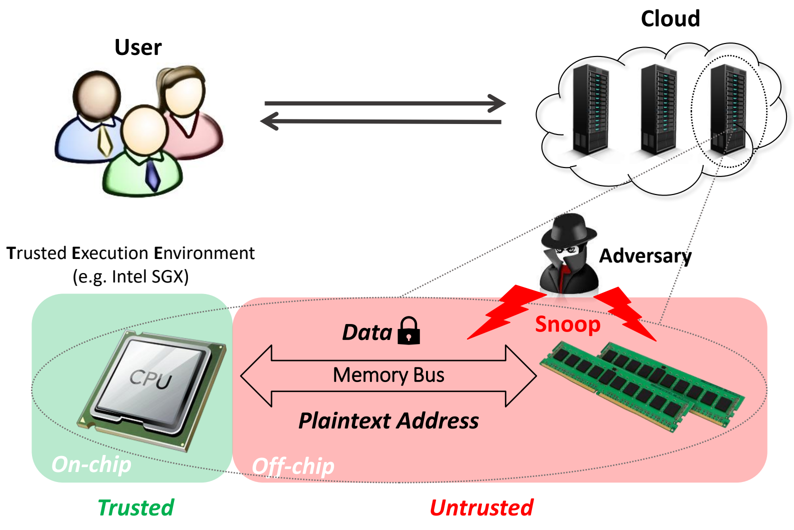


Figure 6: The threat model adopted in the dissertation.

This dissertation assumes 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 [37] 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 [66] and its recent enhancements, as elaborated in the following sections.

2.3 Path ORAM

2.3.1 Operation Basics

Path ORAM is a crypto primitive that protects memory access patterns through obfuscating memory accesses to untrusted external memory [66]. As shown in Figure 7, 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.

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 dissertation. 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$\times$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

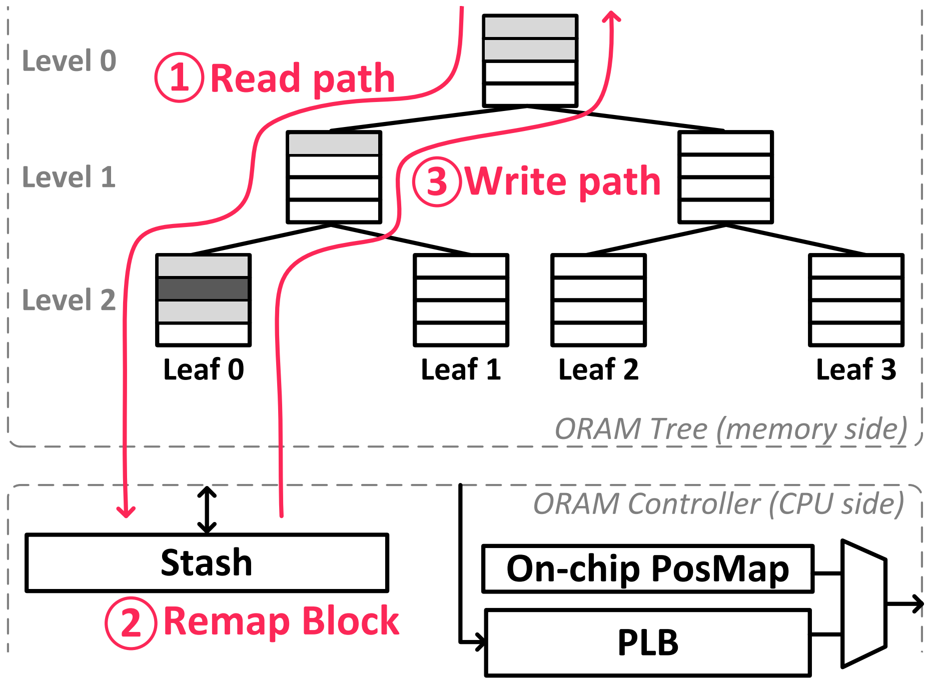


Figure 7: An overview of Path ORAM ($L = 3$, $Z = 4$).

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 [59], resulting in large performance degradation not only to itself but also to co-running applications [72]. In summary, the memory bandwidth consumption of Path ORAM has become the main obstacle that prevents its wide integration in modern computer systems.

2.3.2 Key Enhancements

Many schemes have been proposed to improve the performance of Path ORAM. Next, several closely related enhancements are discussed.

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 [21]. In this dissertation, Freecursive is adopted in the baseline. Three levels of PosMap are constructed — PosMap_1 and PosMap_2 are merged in the main ORAM tree while PosMap_3 is saved completely in an on-chip buffer.

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 [46]. They chose to save the tree top in the stash — they saved up to three levels without incurring considerable stash management overhead. Later studies [72, 49] 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 [49].

To achieve high security, Fletcher *et al.* proposed to defend timing channel attacks by issuing path accesses at fixed rate and pattern [22]. 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 [49]. 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.

Next, the introduction of Ring ORAM optimization will be presented. This optimization slightly alters the Path ORAM protocol to reduce its bandwidth and performance overhead by allocating additional space to the ORAM tree. The following section will detail how it works.

2.3.3 Ring ORAM Optimization

Ring ORAM [58] was developed on top of Path ORAM [66]. It 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 operation that services the memory request of 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 has Z slots. Then, the ORAM tree can hold $Z \times (2^L - 1)$ blocks. Ring ORAM reserves S slots in each bucket, (or $S \times (2^L - 1)$ for the entire tree), for holding dummy blocks only. The remaining Z' slots in each bucket may hold either real data blocks or dummy blocks (Z=Z'+S). Ring ORAM uses around half of Z' space for real data blocks to facilitate random path remapping, similar as that in Path ORAM. Figure 8 depicts Ring ORAM tree organization.

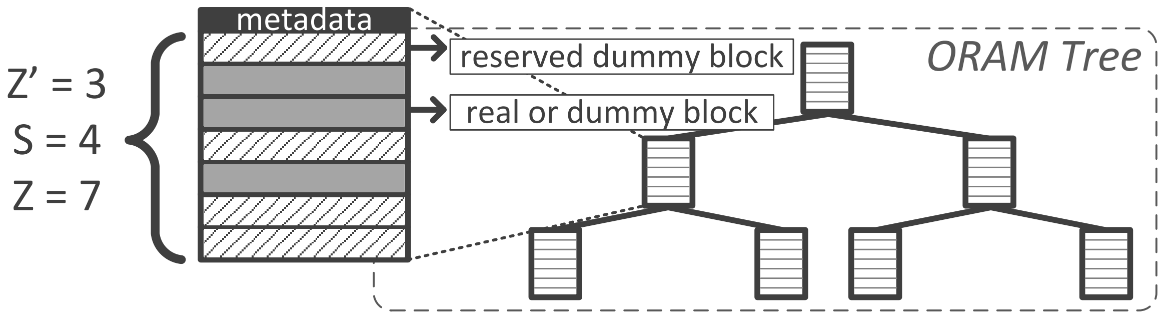


Figure 8: Ring ORAM tree organization ($L = 3$, $Z' = 3$, $S = 4$, and $Z = 7$).

Like Path ORAM, 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 [57, 46]. The path access in Path ORAM, is responsible for servicing the user request, maintaining the tree, and depleting the stash. Ring ORAM, on the other hand, differentiates between online and offline accesses. The former is to service the user program request while the latter refers to maintenance operations. Ring ORAM supports three main operations as follows.

- ***ReadPath:*** this operation is referred to as *online access*, i.e. to service the user program request. To access block A, the ORAM controller first determines its mapped path l and then conducts a two-step access.

  (1) metadata access: The ORAM controller loads the metadata from a separate small tree for all buckets along l. It then identifies the location of block 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 ORAM 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.

ReadPath differs from path accesses in Path ORAM in that it only reads one block per bucket, thereby reducing the memory bandwidth requirement compared to Path ORAM.

- ***EvictPath:*** it 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) writes 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 path access in Path ORAM but requires no specific block to access.
- ***EarlyReshuffle:*** it gets triggered after a *readPath* operation, for a particular bucket if it accumulates S *readPath* operations after the last bucket write. To complete the operation, the ORAM 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). Note that a residue block in the stash might be piggy-backed during a bucket reshuffle.

Since each bucket contains at most Z' real data blocks, *EarlyReshuffle* and *EvictPath* read Z' blocks but write Z blocks. Both operations update the metadata block accordingly.

BackgroundEvict. If the stash is full, Ring ORAM invokes *BackgroundEvict* to prevent the protocol from failing. This is achieved by issuing dummy *ReadPath* that reads a valid dummy block per bucket from a randomly selected path. *BackgroundEvict* ensures the stash occupancy does not increase until the next *EvictPath* arrives. *BackgroundEvict* keeps issuing dummy *ReadPath* operations until sufficient *EvictPath* operations are executed to reduce the stash occupancy below the threshold. *BackgroundEvict* operation converts protocol correctness problem to a performance problem [7, 66].

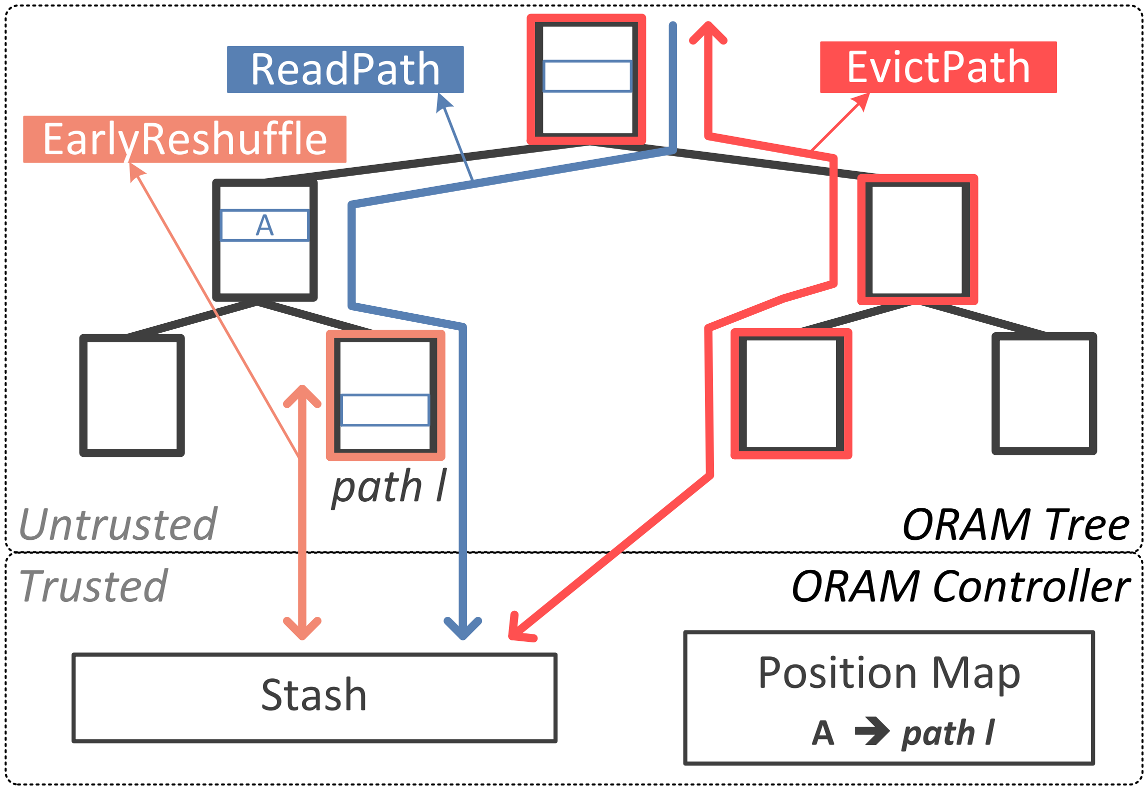


Figure 9: Ring ORAM tree organization and operations with $L = 3$.

Figure 9 depicts an example of a Ring ORAM tree with three levels. The figure illustrates how each Ring ORAM operation affects buckets at each level. In the *EvictPath* operation, all the buckets along the path are affected. In the *EarlyReshuffle* operation, only a bucket with a counter higher than the threshold is affected. In the *ReadPath* operation, exactly one block per level is read into the stash, including the block of interest, as shown in the figure, block A.

Note that *readPath* is considered online access, and services the user program request. Whereas, *evictPath*, and *earlyReshuffle* are offline accesses and responsible for maintaining the tree and depleting the stash.

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

3.1 Intensity Reduction Overview

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.

Path ORAM [66] 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. While Path ORAM reduces the memory access overhead from O(N) in traditional ORAM [25, 26] 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.

- **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.
- **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.
- **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 [22, 20]. 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.

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 [46]. 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 [49]. 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 [82]. Unfortunately, the memory intensity of Path ORAM remains high, which still incurs large performance degradation to the user applications.

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.

3.2 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.

3.2.1 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 $\underline{\mathbf{PT}_p}$, $\underline{\mathbf{PT}_d}$, and $\underline{\mathbf{PT}_m\text{ paths}}$, respectively. To understand their importance, we conduct an experiment to evaluate the frequencies and summarize the results in Figure 10. The experiment settings are in Section 3.4. 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.

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 $4\times$ 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

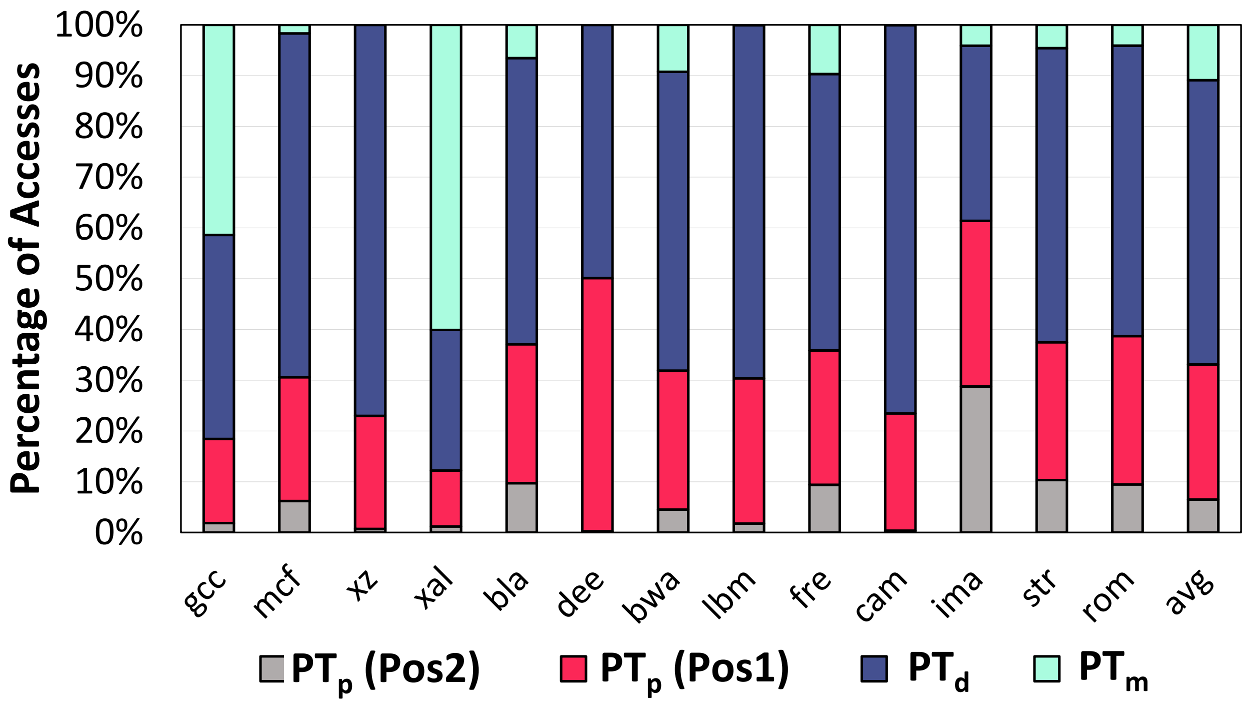


Figure 10: The distribution of different path accesses.

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.

3.2.2 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 [59], i.e., we store around 4GB user data in a 8GB ORAM tree, with all the rest being dummy data blocks.

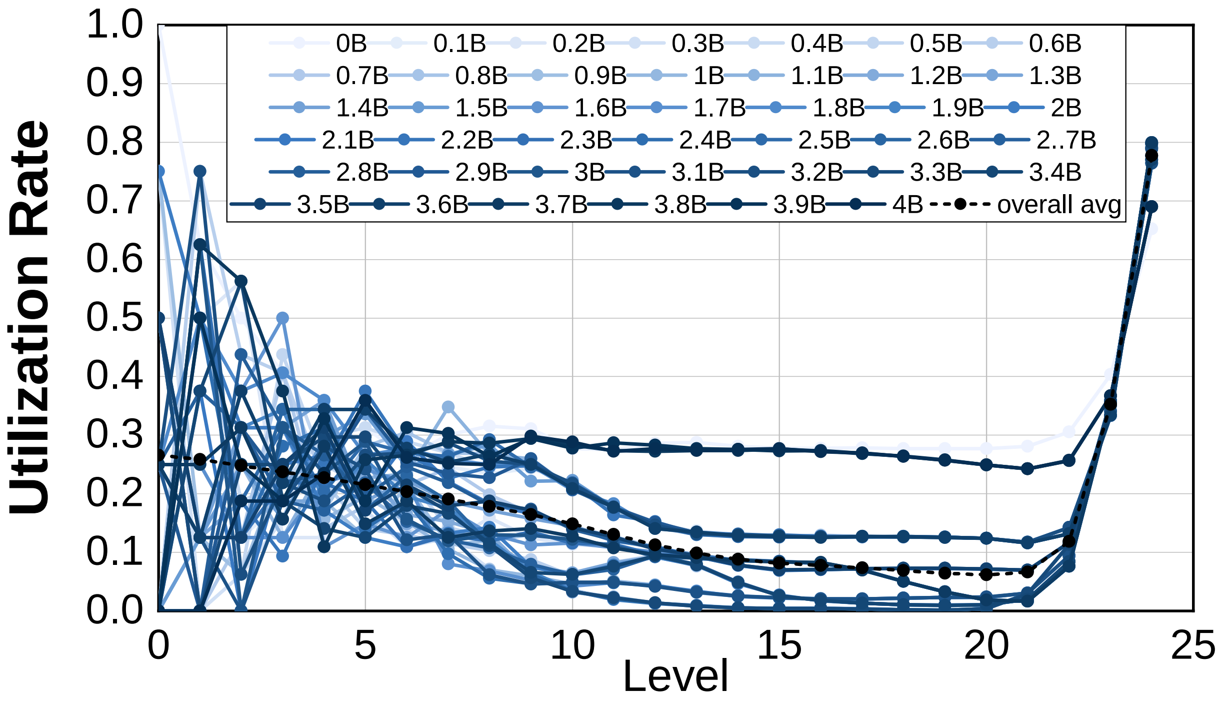


Figure 11: The space utilization at different tree levels.

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. Figure 11 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 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 [66, 59]. 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.

Figure 11 presents the snapshots at different execution points for the typical setting (as shown in the experiment section). While a similar study was reported in [66], 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 Figure 11, 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.

Figure 12 compares the utilization of three different workloads (using the same settings and figure parameters in Figure 11). The results indicate that the utilization trend remains the same for individual workloads.

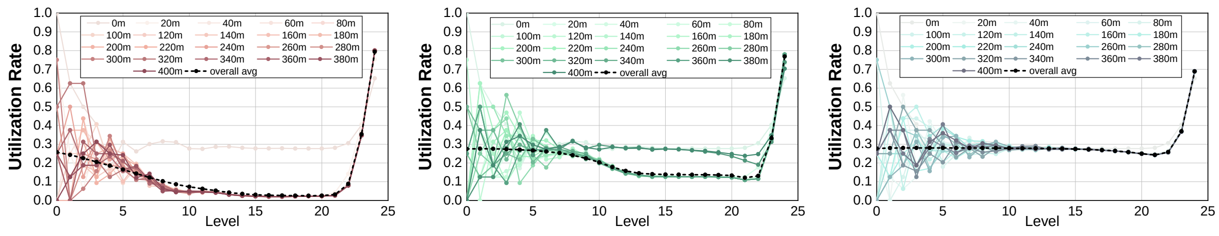


Figure 12: The space utilization behavior per benchmark; gcc (left), lbm (middle), and a random trace (right).

In summary, there exists a significant mismatch of node space utilization, memory bandwidth, and space capacity across different tree levels.

3.2.3 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.

Each path access in Path ORAM has two memory phases — a read phase and a write phase. Once the path ID is determined, in the read phase, we fetch all data blocks along the path and place real blocks in the stash; in the write phase, we search the whole stash, i.e., the blocks from the path and the blocks that were already in the stash, and determine the most appropriate block to be written to the bucket at each. We fill in the buckets from the leaf node to the root.

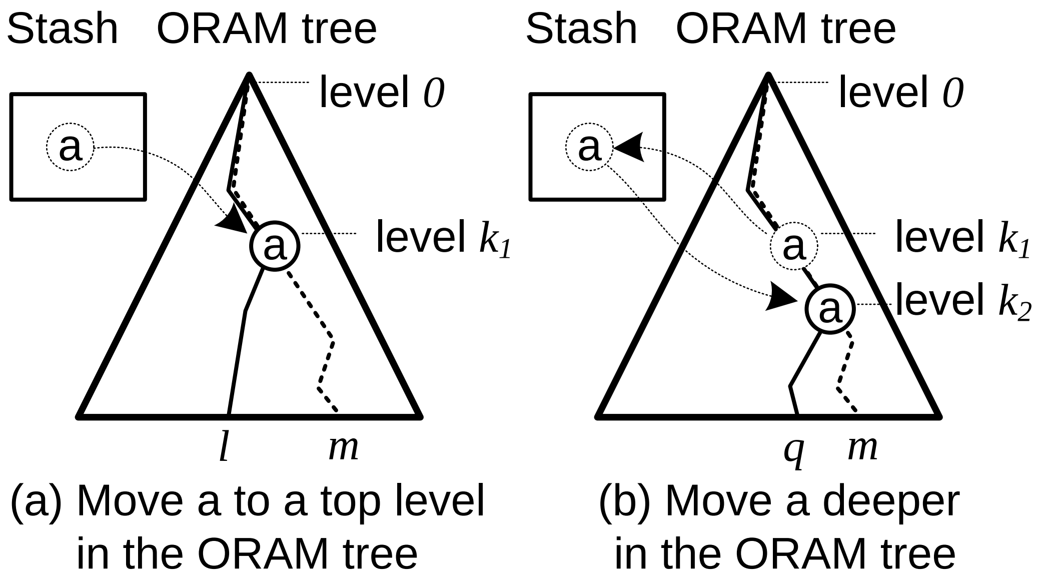


Figure 13: The migration behavior after block a leaves the stash.

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 Figure 13(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.* Figure 13(b) illustrates how it works. Assume a (with path ID m) was written to level k_1 (as in Figure 13(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 $k_1{\le}k_2$. 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. Figure 14 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.

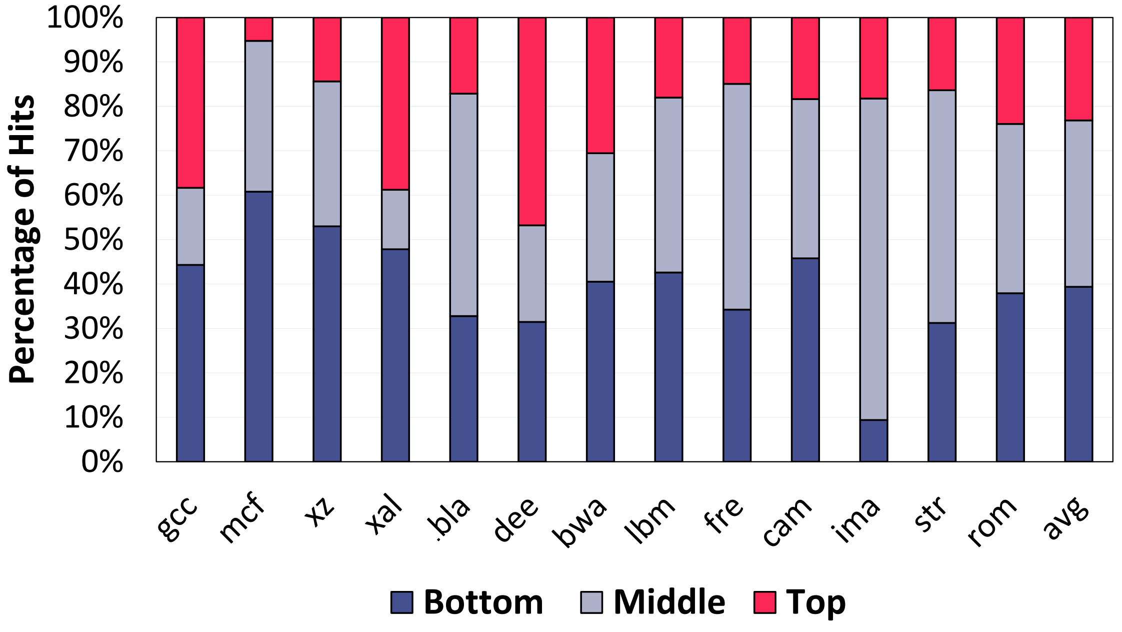


Figure 14: Nodes at top levels have high reuse possibility.

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.

3.3 The IR-ORAM Design

3.3.1 An Overview

In this dissertation, 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.

3.3.2 IR-Alloc: a Utilization-aware Node Size Allocator for Reducing Intensity of $\mathbf{PT}_p$/$\mathbf{PT}_d$ /$\mathbf{PT}_m$ 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 15 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.

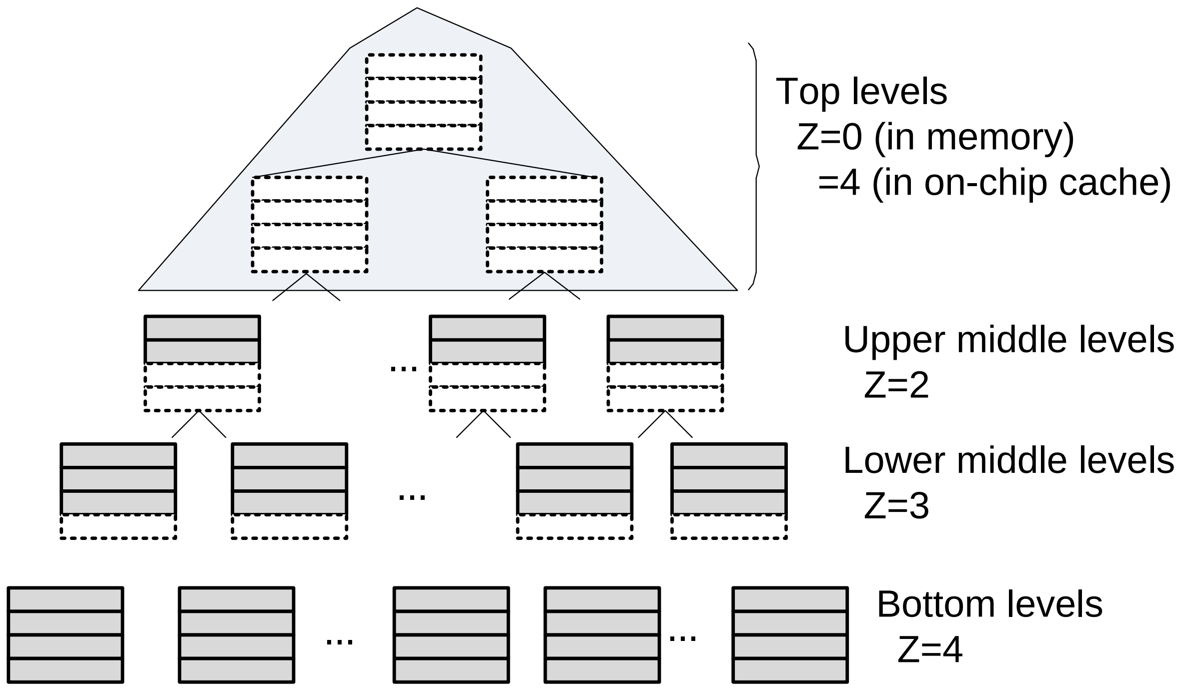


Figure 15: The design of IR-Alloc scheme.

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 Figure 15 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 [66] 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 [59].

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 3.3.5 and study its performance impact in Section 3.4.

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 Figure 11) 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 Figure 11, 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 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.

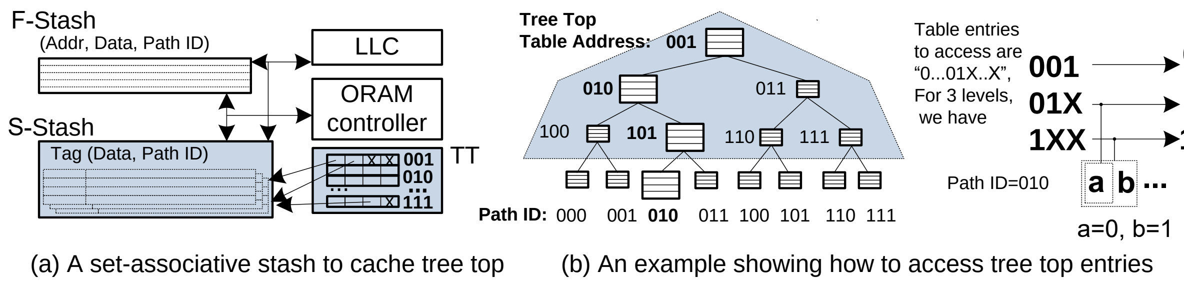

F-Stash
(Addr, Data, Path ID)
LLC
ORAM controller
S-Stash
Tag (Data, Path ID)
TT
001
010
111
(a) A set-associative stash to cache tree top
Tree Top
Table Address: 001
010
011
100
101
110
111
Path ID: 000 001 010 011 100 101 110 111
(b) An example showing how to access tree top entries
Table entries to access are "0...01X..X", For 3 levels, we have
001 → 001
01X → 01a
1XX → 1ab
Path ID=010
a b ...
a=0, b=1

Figure 16: Exploit IR-Stash to effectively cache large tree top.

3.3.3 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 [46]. However, a major issue of this design is its scalability — the stash needs to be expanded to hold (2^m-1)×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 [83, 72]. 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 3.2.3, 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. Figure 16 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 Figure 16(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.

3.3.4 IR-DWB: Converting Dummy Path Accesses for Reducing Intensity of 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.

Figure 17 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 [42] that is also adopted by [67, 74]. Alternatively, candidates can be chosen by maintaining a queue as in eager queue scheme in [42]. The latter approach can be adopted in case hardware overhead is a tight constraint.

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

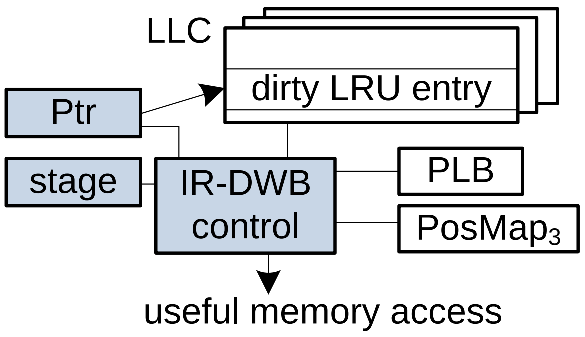


Figure 17: The design of IR-DWB scheme.

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`≠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. The corresponding LLC cache entry is then marked as clean, which improves the replacement performance when it is selected as a victim at a later time. 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 [66, 21, 81]. An LLC-evicted data block, if being dirty, becomes a new memory access. Alternatively, Nagarajan *et al.* [49] proposed a delayed data block remapping policy [49]. 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* [42] 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.

3.3.5 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. Note that here block does not refer to user data block; instead, it refers to the block placeholders in the bucket, also known as slots. 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.

The information about on which path a particular data block resides, and which bucket along the path contains that block, is secret information. The IR-Alloc design does not alter the mapping of user blocks to paths in any way. Thus, just as the block-to-path mapping is protected and secure in Path ORAM, it is also protected and secured in IR-ORAM.

IR-Stash does not alter the operation of the ORAM protocol within the untrusted base. If a block is found in the *S-Stash* lookup, eliminating the need for PosMap accesses, it will not have any observable impact from an outsider's perspective. All paths are indistinguishable, so the absence of a path access cannot be discerned by the attacker, much like when a block is found in the regular stash in Path ORAM.

Similarly, IR-DWB does not affect the indistinguishability of path accesses. It neither alters the pattern nor the frequency of path accesses. Its function is simply to piggyback on existing path accesses. Just as dummy path accesses are indistinguishable from actual data path accesses in Path ORAM, useful early write-back paths are also indistinguishable in IR-DWB. If they were distinguishable, it would be possible to infer the type of path accesses in Path ORAM as well.

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 [59], 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.

3.4 Experiment Methodology

To evaluate IR-ORAM, we used a trace-based simulator USIMM for cycle-accurate DRAM memory simulation [11]. This is similar to the setting that was adopted in recent Path ORAM studies [49, 72]. 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 [21, 82, 49] — 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 [45] 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 [6, 11]. Table 2 lists the L2 misses per kilo instruction (MPKI) for all the benchmarks we picked. Benchmarks were selected from both integer and floating point categories.

Table 1: System configuration for IR-ORAM evaluation.

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 from SPEC and PARSEC suite.

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			

3.5 Experimental Results

We implemented and compared the following schemes.

- `Baseline`: this is the traditional Path ORAM implementation [66] that adopts Freecursive [21] 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 [59].
- `Rho`: it is the ρ design [49] 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 dissertation.
- `IR-DWB`: it implements IR-DWB over `Baseline`.
- `IR-ORAM`: it integrates all three designs. It is built on top of `Baseline`. (It sets Z to 2 and 3 for tree level ranges [10,16] and [17,19], respectively.)

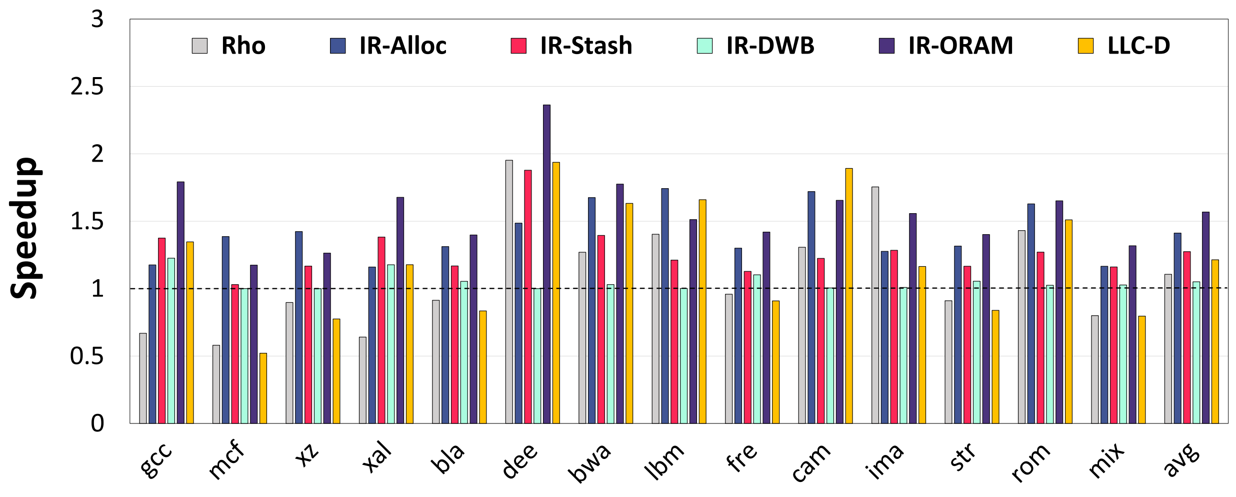


Figure 18: The performance comparison of different schemes.

- `LLC-D`: it adopts the delayed data block remapping policy [49] on top of `Baseline`.
- `IR-Stash+IR-Alloc` (LLC-D as baseline): it is `IR-Alloc` and `IR-Stash` on top of `LLC-D`.

3.5.1 Performance Comparison

Figure 18 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. Figure 19 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.

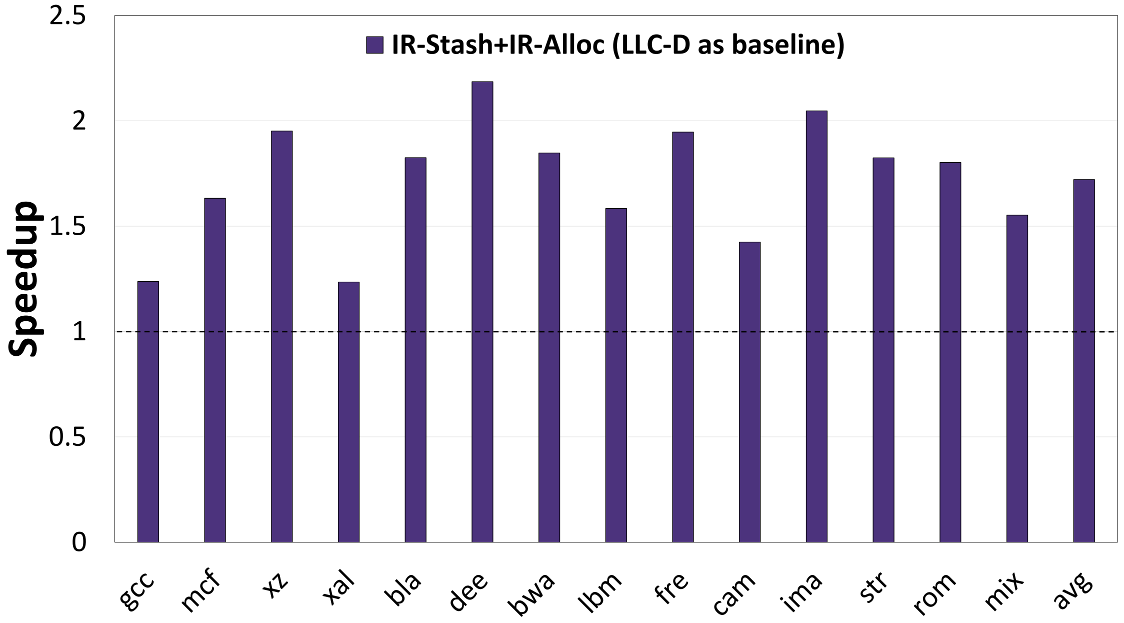


Figure 19: The performance comparison with LLC-D as baseline.

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 Figure 18). 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.

3.5.2 IR-Alloc Overflow

While `IR-Alloc` changes the Z values for two tree level ranges, the discussion in Section 3.3.2 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.

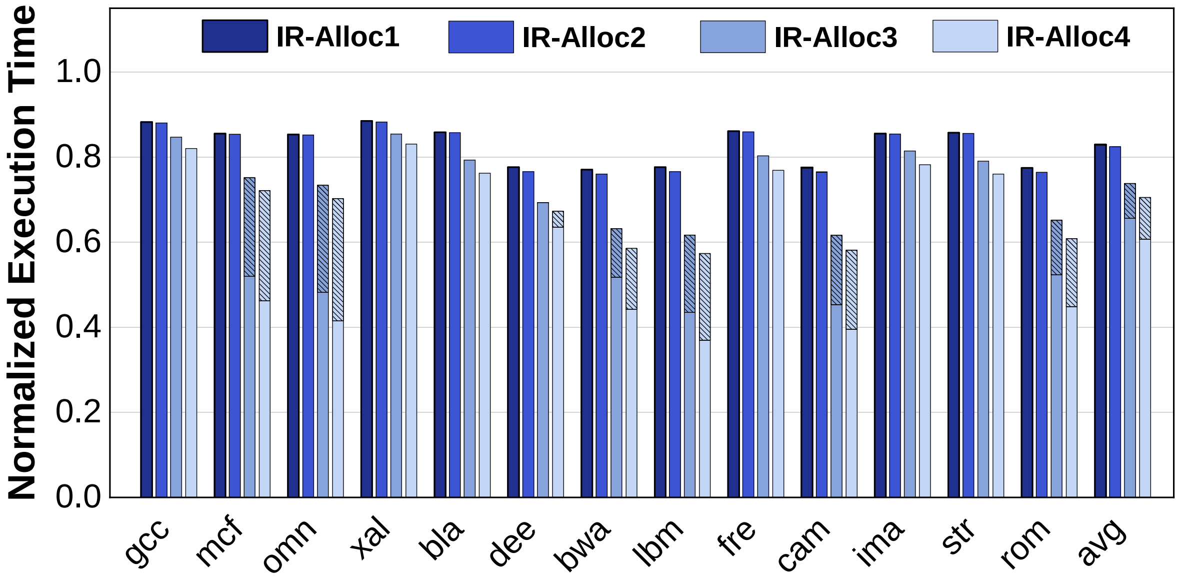


Figure 20: Design exploration of IR-Alloc scheme.

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`
- `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`

Figure 20 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 Figure 18. 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.

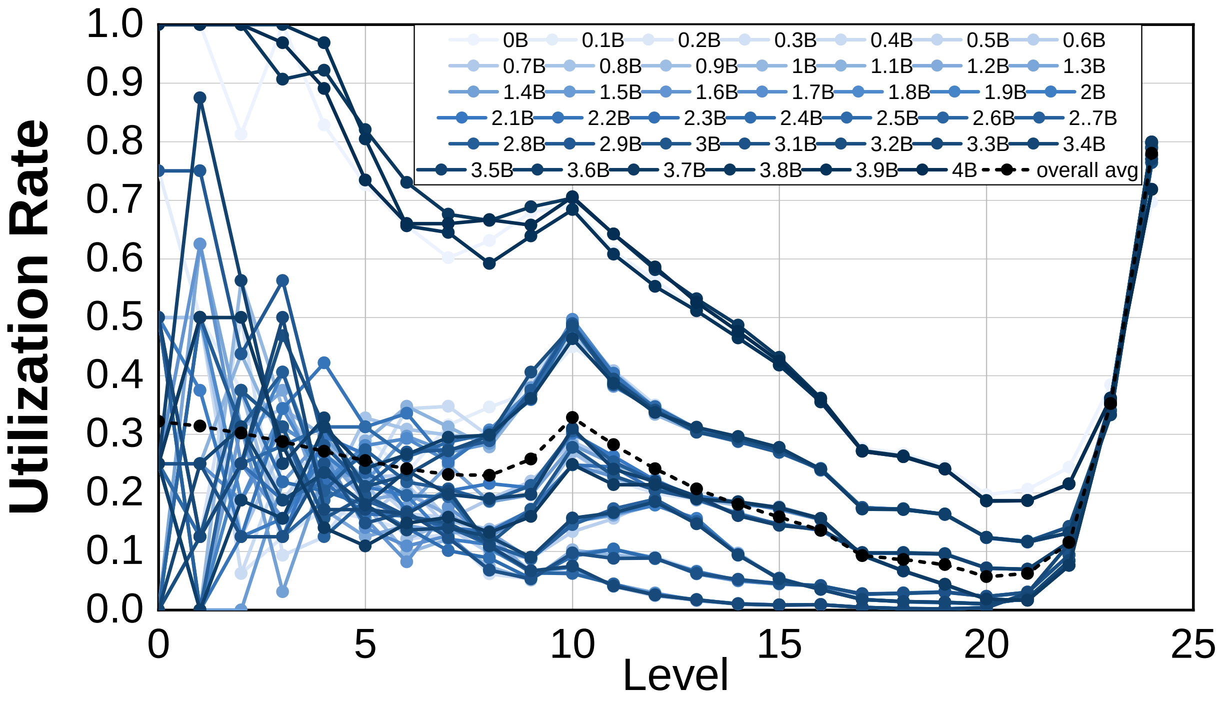


Figure 21: Level utilization with IR-Alloc.

To study background eviction, Figure 21 shows the level utilization experiment similar to that in Section 3.2.2 for `IR-Alloc`. Snapshots were taken at different times of the execution using the same memory trace mix as in Figure 11.

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 Figure 11. 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.

3.5.3 PosMap Reduction

We next investigated the effectiveness of `IR-Stash` in reducing position map accesses. The amount of position map accesses for each program depends on its PLB hit rate and that depends on the program memory access pattern, how much locality it experiences. IR-Stash reduces position map accesses for the blocks that reside in top 10 levels. Figure 14 reports the PosMap access hits across the benchmarks. The benchmarks that have higher hit rates on top levels benefit more from IR-Stash. Figure 22 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`.

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.

3.5.4 Dummy Path Accesses

`IR-DWB` is designed to exploit dummy accesses in timing channel protection mode for early write-backs. Figure 18 shows its speedup. We also investigated its effectiveness in converting dummy accesses. Figure 23 illustrates the percentage of each path access type.

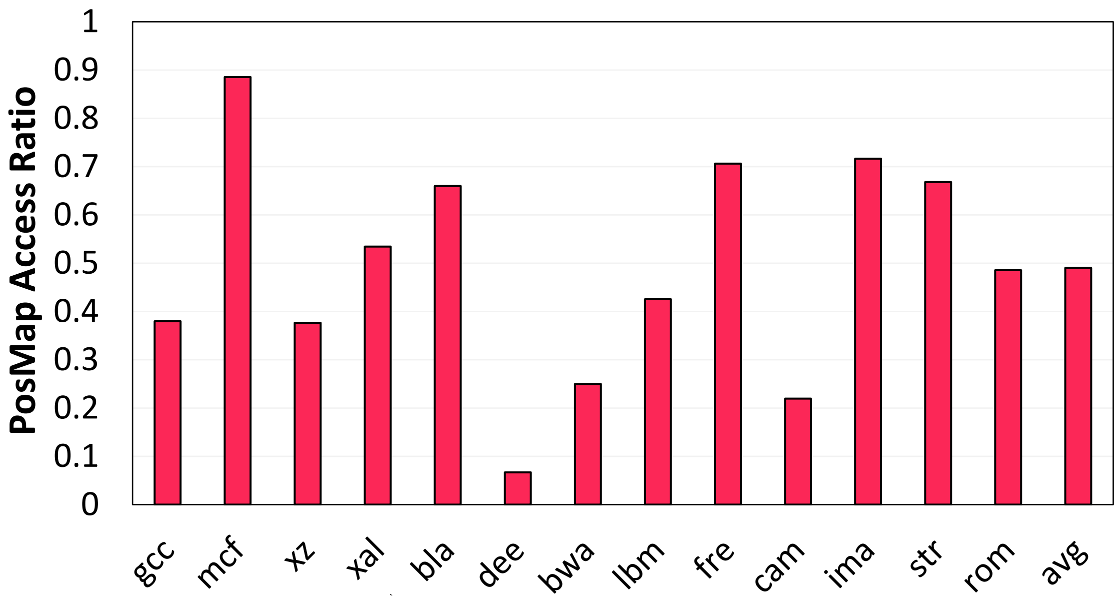


Figure 22: Comparing PosMap accesses with Baseline.

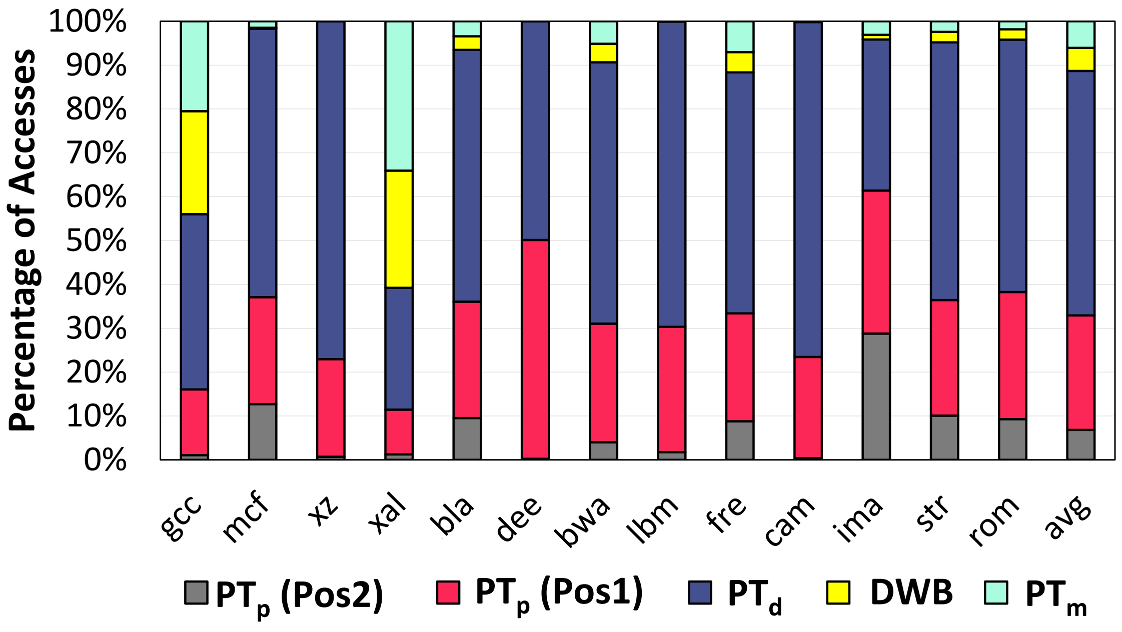


Figure 23: Access type distribution in IR-DWB.

Different benchmarks have a different ratio of dummy accesses. However, as shown in the figure, IR-DWB is able to convert about half of dummy accesses for most of the benchmarks. On average, `IR-DWB` reduces the percentage of dummy accesses from 11% to 6%.

3.5.5 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 Figure 24. 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. Figure 24 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.

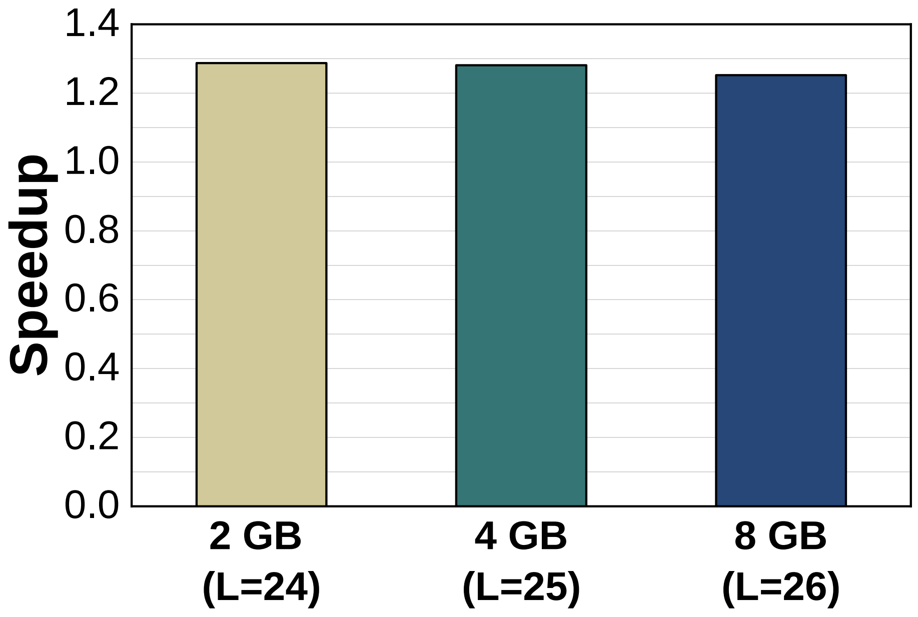


Figure 24: The scalability analysis of IR-Alloc.

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.

3.5.6 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 3.5.2 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 [5]. 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`.

3.5.7 Conclusion

In this chapter, 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

4.1 Space Reduction Overview

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 [37]. Even if the user data may be secured with strong encryption and authentication schemes, e.g., AES encryption [19], Merkle tree authentication [24], it is possible to leak sensitive information from access patterns in memory addresses [86, 78]. To ensure high-level protection of user privacy, it is necessary to adopt an expensive ORAM primitive that obfuscates memory requests from the user program [25, 26].

Ring ORAM is a recently proposed secure primitive for mitigating the large performance degradation of ORAM [58]. Ring ORAM is built on top of Path ORAM [66], 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 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 [66]. This leads to 50% **space utilization**. The space utilization is defined as the size of the real user data over the size of the ORAM tree. Ring ORAM has even lower space utilization as it allocates more dummy blocks. For a typical setting [58], a 12-entry tree bucket keeps (1) five blocks for block remapping. Based on the above discussion, on average, there are 2.5 blocks holding real data while the rest holds dummy data; and (2) seven dummy blocks for Ring ORAM operations. This represents a 2.5/12= 21% space utilization.

Given memory resource is precious for modern computers, 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 [7]. CB prevents the stash overflow possibility with more frequent background eviction, a performance/overhead trade-off optimization proposed for Path ORAM [59]. Unfortunately, space utilization with bucket compaction may be improved to 31%, which remains a major design obstacle for Ring ORAM adoption. For this matter, some studies try to improve the performance of Path ORAM without increasing the space demand. For instance, IR-ORAM is the latest optimization proposed upon Path ORAM that improves the performance while it maintains 50% space utilization. However, it still has lower performance than Ring ORAM. In this work, our goal is to achieve optimum space utilization and performance simultaneously.

This chapter introduces AB-ORAM to improve the space utilization of Ring ORAM implementation while maintaining the high performance benefit of Ring ORAM. The proposed AB-ORAM focuses on **space reduction**. We define space reduction as the reduced total size of the ORAM tree. Note that this reduction only affects the dummy part of the ORAM tree, and the real part, i.e., user data, remains intact. As such, AB-ORAM also effectively improves the space utilization (= user data/ ORAM tree size) as the ORAM tree size is reduced. 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 allo-

cation. AB-ORAM dynamically tracks dead blocks and adaptively allocates them to buckets 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.

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

We use the same threat model as those in prior ORAM studies [66, 58]. Our discussion is based on a standalone secure processor while the design is applicable to cloud setting with a 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 shown that hardware-assisted security enhancements can effectively reduce the encryption and authentication overheads [71, 68, 24]. To prevent memory traces from leaking sensitive information, the baseline configuration adopts Ring ORAM.

4.2 Ring ORAM Space Demand

4.2.1 Ring ORAM with Bucket Compaction

Cao *et al.* proposed to shrink the bucket size while keeping Z' and S parameters intact by introducing the concept of overlap [7]. 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 from the portion dedicated to real blocks 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 the bucket compaction scheme with $Y = 4$, the allocation will be $Z = 8$, $Z' = 5$ and $S = 3$. In this chapter, we built our design on top of this state-of-the-art.

4.2.2 Broad Impact of Space Reduction

Studies have shown that securing memory access patterns demands ORAM [25, 26] as simple obfuscation [86] tends to provide limited security protection. Unfortunately, all ORAM protocols have high performance overheads, in particular, they introduce orders of magnitude more memory accesses. Of different ORAM protocols, Ring ORAM has low online memory bandwidth request, making it a promising protocol for practical deployment.

However, Ring ORAM's space utilization is low, which increases memory contention with co-running applications. As main memory is one of the most precious system resources in modern systems, reducing space demand can effectively make better use of main memory resource, leading to improved system performance, throughput, and power/energy consumption.

4.3 Motivation

For a typical Ring ORAM setting [58], its space utilization is as low as 21%, making it one of the main obstacles that prevent Ring ORAM from wide deployment.

We made two key observations regarding the low utilization of memory space in Ring ORAM. In the following, we elaborate these observations and discuss how to exploit them to improve the space efficiency. The first observation is the existence of dead blocks in the ORAM tree i.e., a memory block, once invalidated by a *readPath* access, remains useless until it gets refreshed by *evictPath* or *earlyReshuffle*. The second observation is that there exists a space/performance trade-off on the bucket size at different tree levels. For the levels close to the leaves, a large S value results in more space consumption but less performance improvement.

4.3.1 Studying Dead Blocks

A bucket slot in Ring ORAM tree can be accessed at most once between any two reshuffles. The ORAM controller marks the slot as *invalid* after its *readPath* but does not reclaim the space until a later *evictPath* or *earlyReshuffle* operation that stores the newly reshuffled, encrypted, and authenticated data. Given *evictPath* adopts reverse-lexicographic order [58] and *earlyReshuffle* is on-demand, the duration of the slot being *invalid* can be relatively long, which lowers down the memory efficiency. In this chapter, the invalid bucket slots are referred to as *dead blocks*.

We conduct an experiment to track the total number of dead blocks for different benchmarks and summarize the results in Figure 25. The settings are listed in Section 4.6. 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. Due to high similarity of the results, the figure only reports the results from three individual benchmarks and the average of all benchmarks. The reason why the results are highly similar is, Ring ORAM is an effective pattern obfuscation protocol, it randomizes the accesses such that different benchmarks have similar access patterns for outside observers.

Dead blocks are generated from online accesses at 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 stabilizes for the rest of program execution. *EarlyReshuffle* always gets triggered when a bucket accumulates S dead blocks. The chance is low at the beginning of the execution. For the path determined by *evictPath*, there are few dead blocks along the path at the beginning of the execution so that only few dead blocks

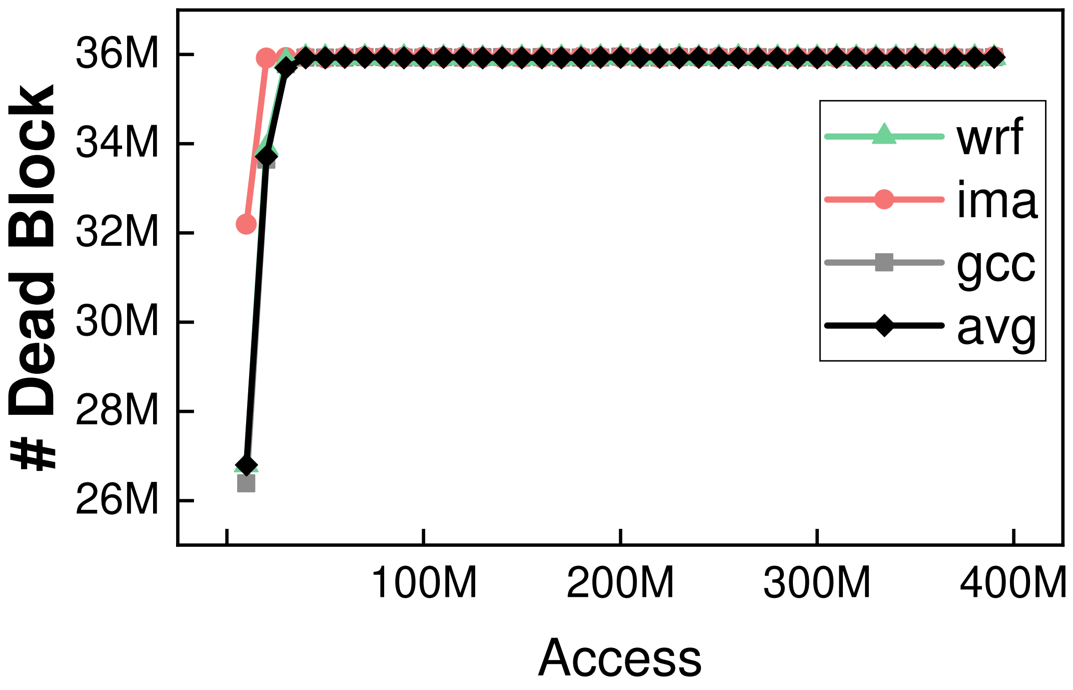


Figure 25: Dead blocks over time for different benchmarks.

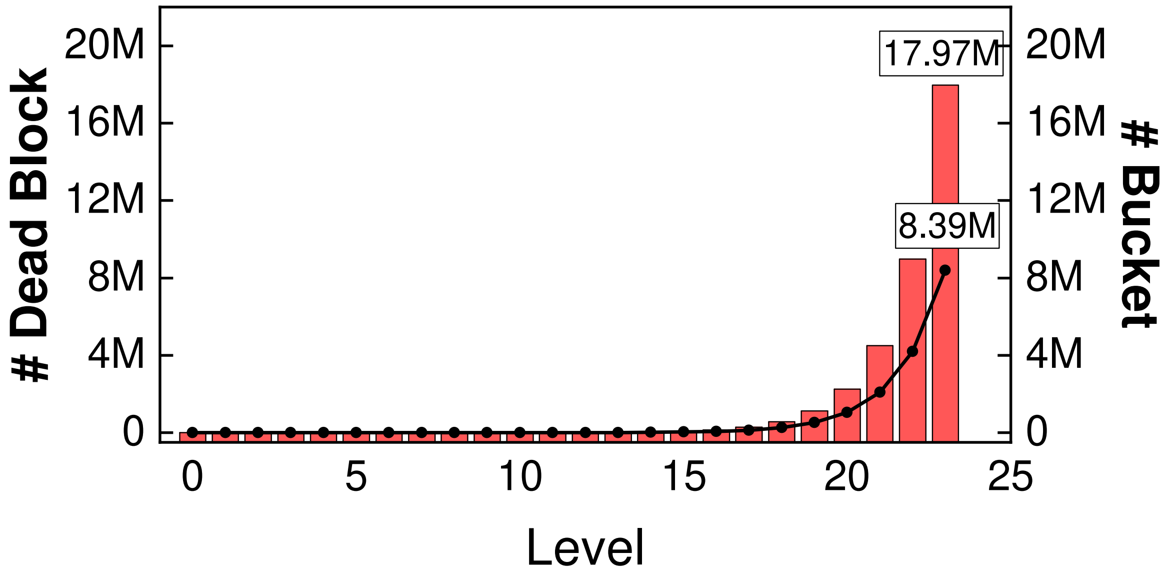


Figure 26: Dead blocks across the levels.

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

For the 24-level ORAM tree in Figure 25, 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.

Figure 26 reports the numbers of existing dead blocks at different tree levels after running 400 million traces. At each tree level (X-axis), the bar shows the number of dead blocks (Y-axis to the left) and the line dot denotes the number of buckets at that level (Y-axis to the right). 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 are 2.1 dead blocks per bucket.

4.3.2 Studying Space/Performance Trade-off

In this section, we study the trade-off between space and performance in Ring ORAM. As discussed before, there are S reserved dummy blocks allocated for each bucket in Ring ORAM. During the *readPath*, only one bucket returns a real block while all other buckets along with the path return dummy blocks. Any bucket, if having been accessed S times, may run out of dummy blocks and thus needs to be reshuffled, i.e., triggering an *earlyReshuffle* on this bucket. Thus, the larger the S value is, the less number of *earlyReshuffle* operations the Ring ORAM would need. Of course, a larger S value results in more space wasted in saving dummy data. In addition, a larger S value leads to more blocks in each path, making it more expensive to complete *evictPath*. Accordingly, if we reduce the S value, the number of *earlyReshuffles* increases but at the same time the cost of *evictPaths* decreases.

We conduct an experiment to expose the space/performance trade-off. In Figure 27, we compare the space and performance impacts when reducing the S values for the bottom seven levels (that account for 99% of the entire capacity). L-x means reducing the S value by three for the last x levels. The baseline adopts the typical setting [58], $Z = 12$, $Z' = 5$, and $S = 7$. From the figure, as we reduce the S values for more levels, the space demand

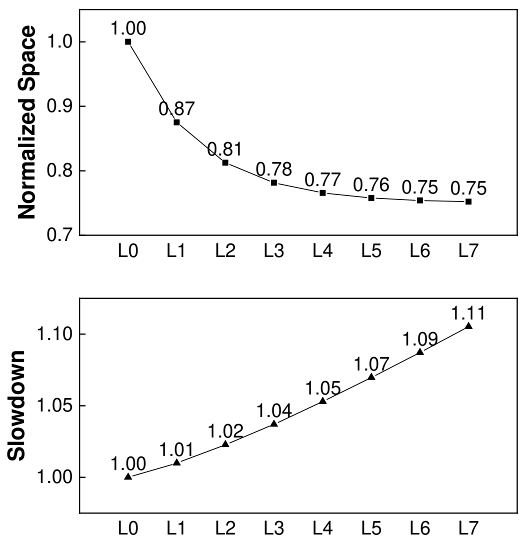


Figure 27: Space demand across different path length normalized to the baseline (top), slowdown over the baseline (bottom).

reduces while the performance degrades. When the bucket size is reduced so is the path length, hence, the path eviction incurs less number of memory accesses. The space saving stabilizes after reducing the last three levels. In the experiment, there is a large increase of the number of *earlyReshuffles*. However, the reduction from conducting cheaper *evictPaths* benefits more. As figure shows, the execution time grows linearly whereas the space reduction is in logarithmic scale. Hence, by shrinking the S value for the levels close to the leaves, we can achieve a significant space reduction while incurring a low performance overhead.

4.4 The AB-ORAM Design

4.4.1 Overview

In this section, we elaborate on AB-ORAM design for reducing the space demand for Ring ORAM. AB-ORAM consists of two designs — one is to early reclaim the dead blocks while the other is to reduce the number of dummy blocks for bottom levels with a non-uniform S value setting.

Dead Block Reclaim. To reclaim dead blocks, we adopt a new block allocation mechanism called remote allocation. To enable remote allocation, we first construct a FIFO queue to track recently generated dead blocks at each bottom tree level and update the metadata accordingly. We then reduce the initial S value, i.e., the number of reserved dummy blocks, for the buckets at bottom levels and extend the S value to its original using the space reclaimed from dead blocks. Given a binary tree doubles its size with every extra level, reducing the S value at the bottom levels can effectively reduce the space demand for Ring ORAM.

Non-uniform S Setting. By exploiting the performance and space-saving trade-off at the bottom levels, we propose to reduce the S value for bottom tree levels. While such a design increases the number of *earlyReshuffle* operations and slightly degrades the performance, it achieves large space savings.

For the first scheme, we need remote allocation and altering S value. While the second scheme only involves altering S value. Thus, we organize the rest of this section as follows. First, we discuss the remote allocation. Then, we discuss how we can exploit altering S value for each of our schemes.

4.4.2 Remote Allocation

4.4.2.1 One Extra Level of Address Mapping

Ring ORAM consists of three levels of address mapping, as shown in Figure 28(a). A memory address of a user request is first mapped to a path ID in the ORAM tree. This mapping is randomized and secured by the ORAM protocol, i.e, encrypted as position map. Given a path ID, we can use the well-known tree organization knowledge to translate it into a list of tree buckets. Depending on the operation, we may get all block addresses for the selected buckets (for *evictPath* and *earlyReshuffle*) or one block address per bucket (for *readPath*). The latter is determined by the metadata, which is also secured and part of the Ring ORAM protocol. Given a tree block address, the OS translates it to the storage location, i.e., the physical addresses in the main memory. From the figure, mapping from a path ID to all its related tree buckets/blocks, and from a tree block to the address in the memory is not protected and is thus known to the attackers.

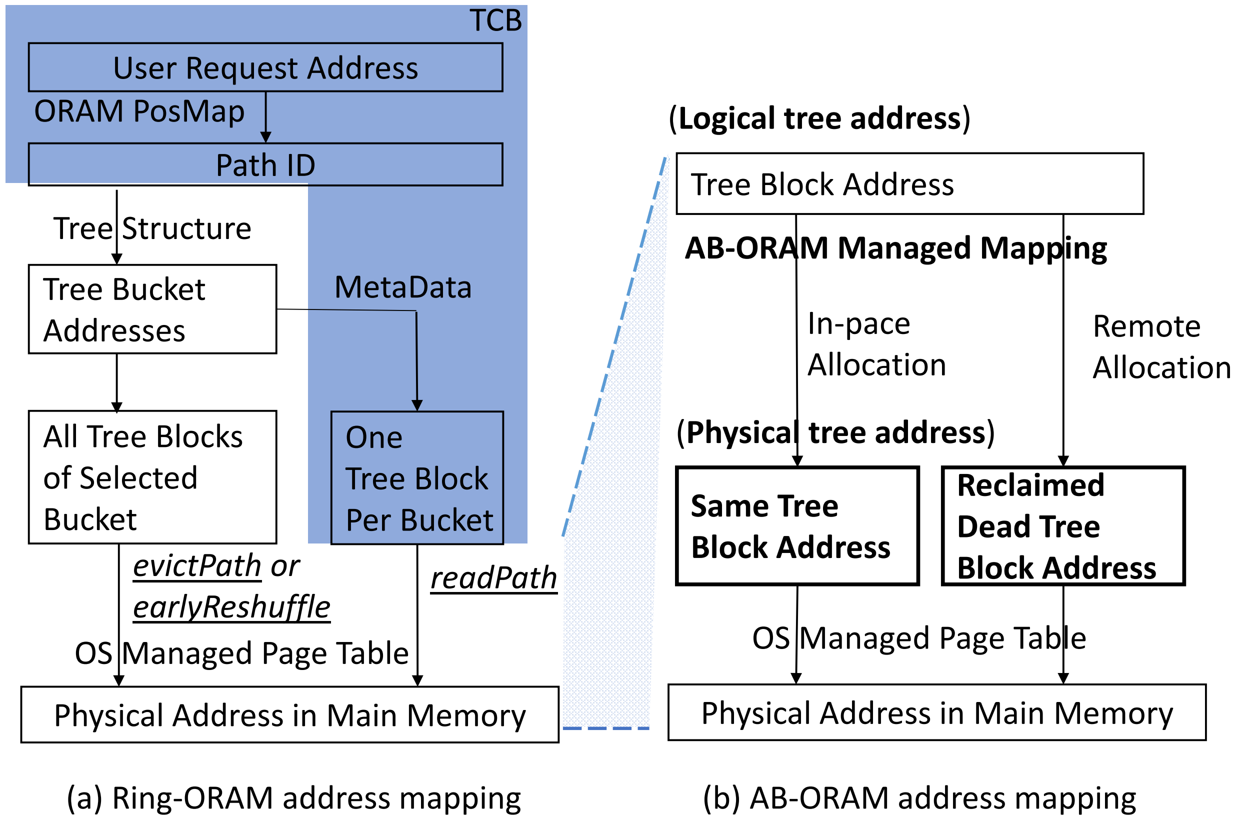


Figure 28: AB-ORAM adds one more level of address mapping in the insecure domain.

To reuse the dead blocks, AB-ORAM introduces one more level of address mapping, as shown in Figure 28(b). The translation from a memory address of a user request to a list of tree block addresses is kept the same as that in the baseline. To facilitate the discussion, we differentiate two tree block addresses — *logical tree addresses* and *physical tree addresses*. Conceptually, the logical tree address of a data block determines where the block should stay according to tree organization. The physical tree address determines its actual location due to space availability. AB-ORAM saves this mapping in bucket metadata, however, the mapping is kept in cleartext and thus known to the attackers.

Based on if a block's physical tree address is the same as its logical tree address, we have two types of data allocation.

- In-place allocation: This refers to the case when a block's physical tree address is the same as its logical tree address. For example, all blocks in the baseline implementation. In AB-ORAM, the block in the top and middle levels keeps the same logical and physical tree addresses.
- Remote allocation: This refers to the case when a block's physical tree address differs from its logical tree address. After fetching a tree block for *readPath*, AB-ORAM marks the corresponding block as dead and may reclaim it by allocating it to a different logical tree block. In Ring ORAM, both *evictPath* and *earlyReshuffle* need to write reshuffled secure data back to the memory. They may demand space allocation and set up the corresponding mapping for remote allocation. The mapping is kept in the metadata in cleartext.

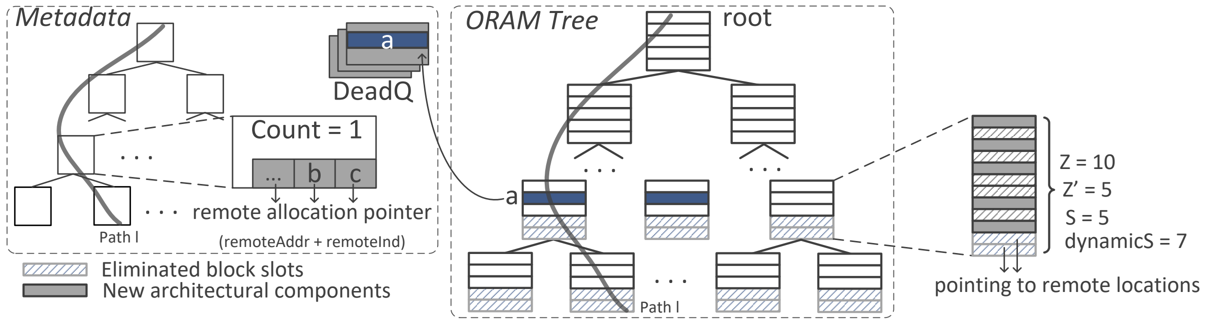


Figure 29: An overview of remote allocation in AB-ORAM.

4.4.2.2 Tracking Dead Blocks

To enable remote allocation, we need to dynamically identify the dead blocks in the ORAM tree so that we can reuse them later. This consists of two sub-tasks. One is to collect the addresses of dead blocks while the other is to mark the status of tree blocks in metadata.

Tracking Queues. In Ring ORAM, dead blocks are generated from *readPath* operations — each *readPath* fetches L blocks such that all of them become *dead* after the access. While all L blocks can be potentially claimed, our study in Section 4.7.4 reveals that the dead blocks in the top and middle levels tend to have short lifetimes and they account for a small portion of the total memory space. As such, we skip these levels and maintain several FIFO queues with one for each bottom level, referred to as *DeadQ* queues, to track dead blocks at each corresponding level. Each entry in the queue maintains two fields that define the physical location of a dead block (or an empty slot): {`slotAddr`, `slotInd`}.

AB-ORAM maintains all the *DeadQ* queues inside the processor, i.e., on-chip. However, this information does not need to be secured. Our security analysis shall prove that it does not compromise security protection if the attacker knows this information. Given the limited on-chip space, each *DeadQ* maintains a small number of entries, e.g., 1000 entries. Since they are FIFO queues, the maintenance cost is low. The design goal of the *DeadQ* is not to track all dead blocks at each level. Instead, it is to collect a good amount of dead blocks so that we can exploit dead blocks to meet part of the space allocation demands for the following *evictPath* and *earlyReshuffle* operations. We keep one queue for each level because according to our analysis the lifetime of dead blocks from different levels exhibit orders of magnitude difference as shown in Figure 35.

Tracking Metadata. Since we add one level of address mapping, we need to precisely know if a tree block adopts in-place allocation or remote allocation. AB-ORAM achieves this by keeping per block status information in the metadata of each bucket. 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. Whereas the slot-related metadata indicates information about the slot itself, i.e. the physical location.

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.

AB-ORAM adds five pieces of metadata to Ring ORAM: four block-related (`remote`, `remoteAddr`, `remoteInd`, and `dynamicS`) and one slot-related (`status`). The `remote` flag

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

	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 ID of 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 location
	`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 of the remotely allocated 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`)

indicates whether the corresponding block adopts in-place or remote allocation, i.e., if the logical and physical locations of the block are the same. In the case that they are not the same, `remoteAddr` and `remoteInd` identify the physical location of the block in the tree.

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 *DeadQ*.

Table 3 states the size of metadata fields in terms of ORAM parameter. Note that 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 match those introduced in Section 2.3.3.

Tracking procedures. In summary, we have the following two lightweight procedures.

- `markDEAD()`: it marks the `status` of a slot as `DEAD` when its occupant block turns dead (i.e. `valid` $= 0$). Note that the block may be either in-place-allocated or remote-allocated. It is invoked at the metadata access of *readPath*.
- `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 of *readPath*.

4.4.2.3 Remote Allocation

In Ring ORAM, only *evictPath* and *earlyReshuffle* can write data blocks to the main memory. All data blocks take in-place allocation in the baseline Ring ORAM implementation.

Next, we illustrate how to remotely allocate a data block (with logical tree address A) at a bottom tree level.

(1). We first dequeue a *DEAD* block from its corresponding *DeadQ*. Let us assume the tree block address of it is T.

(2). We then write block A in block T. Note that no access or update on metadata for block T is needed. Since block T is looked up from the *DeadQ*, its `status` has already been updated as `ALLOCATED` (at the time it was queued).

(3). We update the block-related metadata of block A. The `remote` flag is raised and `remoteAddr` and `remoteInd` are set to point to block T. All other block-related metadata pieces are updated as they are in the Ring ORAM.

Figure 29, demonstrates the remote allocation. In the figure, after accessing block a on path l, it is marked as dead and added to the *DeadQ*. Block b and c are remotely allocated and the mapping identifies the physical tree addresses.

4.4.3 Altering the S Value for Space Savings

We next elaborate on how to change the S value for better space savings. It consists of two designs. One is to extend the S value to take advantage of the remote allocation. The other is to set the non-uniform S values for more space savings.

4.4.3.1 Extending the S Value with Remote Allocation

While remote allocation can early reclaim the space occupied by dead blocks, the baseline tree allocation actually cannot exploit its benefit — in the baseline, every logical tree block has its physical tree block allocated so that there is no need to "reuse" the space of dead blocks.

There exist two alternative strategies to exploit remote allocation. Based on the typical ORAM setting $Z' = 5$, $S = 7$, $Z = 12$, $A = 5$, assume we apply remote allocation to the bottom levels, taking level 23 as an example.

(1). We allocate 12 blocks ($Z = 5 + 7 = 12$, $S = 7$) for each bucket at this level. At runtime, based on the generation rate of dead blocks, we can extend the bucket size to $Z = 5 + 9 = 14$, $S = 9$, i.e., each bucket can sustain 9 *readPath* accesses, instead of 7 *readPath* accesses, before triggering an *earlyReshuffle*.

(2). We allocate 10 blocks ($Z = 5 + 5 = 10$, $S = 5$) for each bucket at this level. At runtime, we recover the bucket size to $Z = 5 + 7 = 12$, $S = 7$. Each bucket can still sustain 7 *readPath* accesses, even though we physically only allocate 5 reserved dummy blocks.

For either strategy, after extending the S value, each bucket has two fewer physical blocks, e.g., strategy (1) allocates 12 entries for one bucket but tries to use it as a 14-entry bucket. We, therefore, allocate two logical blocks to the reclaimed dead blocks. The two strategies have different emphases — the first strategy saves no space comparing the baseline. However, it reduces the number of *earlyReshuffle* operations and thus potentially improves the performance. The second strategy focuses on space savings. It uses smaller space for the initial tree allocation but tries to achieve the same effectiveness as the baseline. Given this chapter focuses on space savings, we adopt the second strategy in AB-ORAM.

AB-ORAM initializes the ORAM tree with the bucket size for bottom tree levels being set as $Z - r$ where r denotes the applied space reduction. Based on the study in Section 4.3, we identify r as 2 for the baseline setting. The r value depends on the choices of Z, Z', and S values, i.e, it depends on the ORAM tree, while being independent of the secure applications. After one complete round of tree reshuffle, i.e, applying *evictPath* to every path, we start to extend S values for the bottom levels. The number of sustained *readPath* accesses before *earlyReshuffle* is also updated to the new value.

In the design, when the *DeadQ* is empty, we may skip extending the S value for a bucket at the bottom level. This may happen either at *evictPath* or *earlyReshuffle* time. To facilitate the transition and the empty-queue case, we keep a counter `dynamicS` for each bucket, as in Table 3. It tracks the number of *readPath* that it can sustain, `dynamicS` is extended to $S + 2$

only for the buckets that allocate their two logical tree blocks in reclaimed dead blocks.

4.4.3.2 The Non-uniform S Value

The motivational study in Section 4.3 revealed that, if we choose to reduce the S value for a number of bottom tree levels, there exists a trade-off between performance overhead and space savings. Therefore, we can take a non-uniform S value design that sets S_1 and S ($0 \leq S_1 < S$) values for the bottom k tree levels and for the other tree levels, respectively. Here S is the same as the baseline while S_1 is a smaller value. By reducing the S values for k bottom levels, we tend to suffer from a small performance degradation but achieve space savings.

Figure 27 in Section 4.3 shows that the space savings are about to saturate at $L3$ while the performance loss is low (about 4%). The result was for $S_1 = S - 3$. However, if we take a different S_1 value, the trend of space savings stay the same while the performance loss may vary. We, therefore, set $k = 3$ or $k = 2$ and adjust S_1 according to the space utilization. Our experiment study shows that S_1 can be set as S-2 and S-1 for the baseline Ring ORAM and an optimized implementation that makes better use of space, respectively.

We propose two designs in the paper — (1) Initially allocating smaller buckets (with smaller S value) for several bottom tree levels and extending the S value at runtime; (2) Assigning a smaller S value for several bottom tree levels. While both designs shrink the S value, they differ from each other as they have different design goals. The former allocates fewer physical tree blocks for these buckets and exploits dead blocks to mitigate the physical space reduction, i.e., recover to the same S value and sustain the same number of *readPath* accesses as those of the baseline. The performance overhead comes from accessing remote blocks for a subset of tree entries. The latter permanently reduces the S value for the bottom tree levels such that these buckets can sustain fewer *readPath* accesses. The performance overhead comes from increased *earlyReshuffle* operations. By shrinking the S value, the latter design improves the *evictPath* performance.

It is worth noting that the Skewed Merkle Tree was proposed to reduce the number of memory accesses [69, 87]. A skewed tree features imbalanced left and right subtrees, resulting in varying path lengths. Conversely, AB-ORAM maintains the full binary tree structure but reduces the size of the nodes at certain levels, meaning that paths retain the same length.

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

	Ring ORAM [58]	IR-ORAM [56]	Bucket Compaction [7]	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	-	-

4.4.4 Comparison to the State-of-the-arts

Table 4 summarizes the latest optimizations proposed on ORAM. 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 [56]. 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 [66] while the principal can be adopted to optimize the portion of Z' entries of tree buckets in Ring ORAM.

Bucket Compaction (CB) [7] 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. 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 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 chapter, i) extending the S value for bottom levels by reclaiming dead blocks, and ii) setting non-uniform S values for the ORAM tree. For the former, 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 latter design shrinks the bucket size of levels close to the leaves, which helps to reduce the overhead of each *evictPath* operation. This increases the number of *earlyReshuffle* operations 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-ORAM or CB, We are able to further reduce the space demand of Ring ORAM. The space utilization of the fully optimized Ring ORAM implementation is made comparable to that of Path ORAM, i.e., around 50%.

4.5 Security and Correctness

In this section, we show that AB-ORAM ensures the same level of security guarantee as that of the baseline Ring ORAM. Since AB-ORAM consists of two design, we next elaborate that both designs are secure.

4.5.1 Remote Allocation is Secure

To prove remote allocation is secure, we refer to the address mapping enhancement in Figure 28 in Section 4.4.2.1 The key observation is that the added address mapping is outside of the secure domain, that is, we utilize public knowledge to improve space utilization and leak no secure information.

To utilize the space of dead blocks, AB-ORAM tracks the generation of dead blocks, early reclaims dead blocks, queries the status of selected blocks, and shrinks/extends the S value for the bottom tree levels. Tracking the generation of dead blocks leaks no secure information as it is public knowledge — attackers, without performing AB-ORAM, can conduct the same information collection, i.e., the blocks accessed by *readPath* become dead. We maintain a queue for each level and enqueue/dequeue in cleartext. Given a dead block, it is well-known if it is queued, or skipped as the queue is full. Collecting such information reveals no secure information.

Reclaiming dead blocks leaks no secure information as, to reclaim dead blocks, we dequeue them from *DeadQ* and exploit them as a buffer to store the data of corresponding logical tree blocks. The mapping is known to the public. Such an extra mapping is secure because if it is not, we can construct a simple attack to the baseline Ring ORAM. Conceptually, the decision on choosing a particular logical tree block from a bucket at *readPath* is secure. The address mapping introduced in AB-ORAM does not change the above decision and kicks in only when we know the address of a logical tree block.

Querying the status of data blocks is secure as the query is integrated with the metadata access in the baseline. The metadata access is performed before each *readPath* and thus AB-ORAM does not introduce extra protocol access steps.

In the same way that real and dummy blocks are indistinguishable in Ring ORAM, their dead and reused versions are indistinguishable in AB-ORAM. Therefore, an attacker cannot infer anything about a block being real or dummy by collecting a dictionary of all remote mappings in AB-ORAM. Also, the temporal locality would not affect this matter either because if it would, one could guess the type of the accessed block based on its location along the path in Ring ORAM.

In summary, the remote allocation design in AB-ORAM is secure and leaks no secure information.

4.5.2 Altering the S Value is Secure

According to the security analysis in the baseline Ring ORAM [58], the three types of operations can be divided into two groups: (1) user-application-dependent *readPath*; (2) maintenance-oriented *earlyReshuffle*, and *evictPath*. Ring ORAM ensures all *readPath* accesses are indistinguishable, while the knowledge about when to perform maintenance operations is well-known and leaks no secure information. We next prove that, by altering the S value (in dead block reclaim and non-uniform S design), these operations remain secure.

For *readPath*, altering the S value has no impact on Ring ORAM mapping (i.e., creating/accessing PosMap entries). Consequently, all *readPaths* remain indistinguishable — each of them initiates a metadata access and then fetches one block from each bucket along the path. Each bucket contains at least one dummy slot to support one more *readPath*. Therefore, no secure information leaks from collecting *readPath* operations.

For *earlyReshuffle*, it is public knowledge about when to be triggered on a particular bucket. Altering S is advertised as cleartext `dynamicS` such that the same strategy is applied to trigger *earlyReshuffle*, which leaks no secure information.

For *evictPath*, it is triggered at a fixed interval, i.e., once for every five *readPath* accesses and uses the fixed reverse-lexicographic order, which leaks no secure information.

For the special case discussed in Section 4.4.3.1, i.e., when *DeadQ* is empty, we may skip extending the S value, i.e., allocating $Z = 5+5 = 10$ entries instead of $Z = 5+7 = 12$ entries for a bucket at the bottom level. This results in assigning `dynamicS`$= 5$ for this bucket. This is secure as this is public knowledge, similar to assigning a cleartext $S = 7$ in the baseline. The slight security difference here is, now an attacker guesses if any of the first 5 accesses to a 10-entry bucket contains real data (instead of guessing from 7 accesses to a 12-entry bucket). For the non-uniform S value design, the security indication is, assume we reduce S to 3, an attacker guesses if any of the first 3 accesses to an 8-entry bucket contains real data. Given this guess needs to be combined with all bucket accesses along the path, i.e., only one bucket returns the real data, this security difference is negligible as we demonstrate empirically in the following experiment.

4.5.3 Empirical Security Analysis

We set up an experiment to demonstrate how AB-ORAM preserves the same security guarantee as Ring ORAM. We simulate one billion traces of 17 benchmarks in this experiment. All other configurations are listed in Section 4.6. We measured the success rate of an attacker guessing the real block during *readPath*. The attacker guesses one block out of L

blocks randomly. Figure 30 indicates the success rate, i.e., the number of correct guesses by the attacker over the total number of *readPaths*. On average, the baseline exhibits a success rate of 0.041665, while for AB-ORAM, it is 0.041670. As shown in the figure, AB-ORAM closely follows the baseline. As one would expect, the success rate for all applications is around 0.041666 (= 1/24), which highlights that path accesses are indistinguishable in Ring ORAM, which is preserved by AB-ORAM as well.

This result is expected because AB-ORAM does not alter the fundamental nature of the *readPath* operation. Similar to the Ring ORAM baseline, where only one block is read per *readPath* operation, in AB-ORAM, this principle persists. Out of the 24 accessed blocks, only one block contains real data; the rest are dummy blocks. Regardless of the number of reserved dummy blocks (i.e., the allocated S value) a bucket has, it must have at least one reserved dummy block available per bucket along the path at the time of *readPath*. This principle remains consistent in AB-ORAM as it does in Ring ORAM. As mentioned before, the only slight difference is that the attacker is guessing from fewer accesses to a smaller bucket size. Our empirical experiments confirm that this difference is negligible.

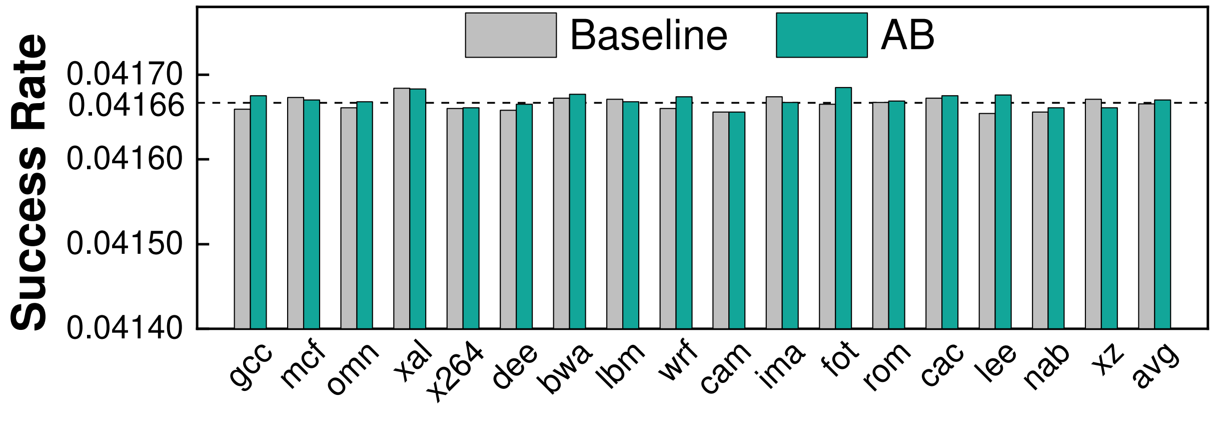


Figure 30: Empirical study on AB-ORAM security implication.

4.5.4 Correctness

Ring ORAM follows Path ORAM and fails if a data block is mapped to a path that contains no dummy entry for the Z' portion. Given AB-ORAM does not change the Z'

value for the ORAM tree, AB-ORAM can save the same number of data blocks in each bucket. The address mapping from the user address to the tree path remains the same. AB-ORAM fetches one real data block from one path, the same as the baseline. As such, AB-ORAM introduces no correctness issue.

4.6 Experimental Methodology

To evaluate AB-ORAM, we used USIMM [11], a widely-adopted trace-driven cycle-accurate DRAM simulator in the literature to evaluate ORAM schemes. Table 5 lists the configuration details. We used the Pin tool [15] to collect traces from 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 results remained stable. Table 6 lists the benchmarks and their LLC misses per kilo instruction (MPKI) in the experiments.

We modeled a Ring ORAM tree with 24 levels, $Z = 12$, and $Z' = 5$, $S = 7$. We integrated Bucket Compaction [7] in the baseline, as set $Y = 4$, $Z = 8$, $Z' = 5$, and $S = 3$, i.e., the tree occupies $(2^{24} - 1) \times 8 \times 64\text{B} = 8\text{GB}$ memory space. Following the prior work [66, 81, 82, 49, 7, 56], the protected user data occupies around 50% of all Z' entries in buckets, that is, $2^{(24-1)} \times 5 \times 50\% \times 64\text{B} = 2.5\text{GB}$. We adopted a tree cache that saves the top 10 levels on-chip.

We implement and evaluate the following schemes.

- `Baseline`: It implements Ring ORAM [58] and integrates Bucket Compaction [7], with $Y = 4$, $Z = 8$, $Z' = 5$, and $S = 3$. Note, all other schemes are built on top of this.
- `IR`: It implements IR-ORAM utilization optimization IR-Alloc in [56]. It sets $Z' = 4$ for levels in range [L10, L18], and $Y = 3$.
- `DR` (<u>D</u>ead Block <u>R</u>eclaim): It sets the bucket size to $Z = 6$ ($Z' = 5$, $S = 1$) for [L18, L23], then extends S value by 2 via remote allocation.
- `NS` (<u>N</u>on-uniform $\underline{S}$): It sets the bucket size to $Z = 6$ ($Z' = 5$, $S = 1$) for [L22, L23].

- `AB`: It combines `DR` and `NS`. It sets $Z = 6$ ($Z' = 5$, $S = 1$) for [L18, L20], and $Z = 5$ ($Z' = 5$, $S = 0$) for [L21, L23].

Table 5: System configuration for AB-ORAM evaluation.

Processor Configuration	
Processor Fetch Width/ ROB Size	4 / 256
Memory Channels	4
DRAM Clk Frequency	800 MHz
L1 / L2 D-cache	4-way 64KB / 8-way 256KB
L3 cache (LLC)	16-way 2MB
ORAM Configuration	
ORAM tree levels	24
Bucket size/ Block size	12 / 64B
Stash entries	300
Dedicated tree top cache	256KB (4K entries)
On-chip PLB / PosMap	64KB / 512KB

Table 6: Evaluated benchmarks of SPEC suite.

Integer Benchmark	read MPKI	write MPKI	Float Benchmark	read MPKI	write MPKI
gcc	0.1	0.5	bwa	0.0	4.1
mcf	28.2	0.2	lbm	0	15.3
omn	0.3	0.06	wrf	0.1	1.0
xal	0.1	0.2	cam	0.0	7.1
x264	1.6	2.1	ima	0.2	2.1
dee	0.0	14.7	fot	0.03	1.56
xz	0	15.5	rom	0.0	13.7
lee	0.01	0.01	nab	0.1	0.2
			cac	0.0	5.4

4.7 Experimental Results

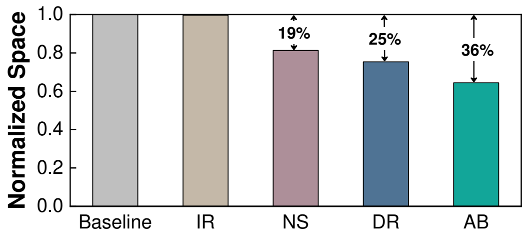
In this section, we discuss the result of evaluated schemes described in Section 4.6.

4.7.1 Main Results of Space and Performance

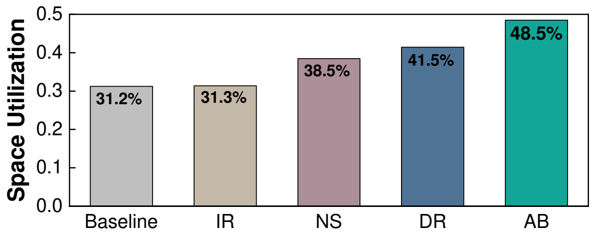
Figure 31 presents the main result of our evaluation. Figure 31a reports the total space consumption of different schemes normalized over `Baseline`. Figure 31b shows the space utilization. Figure 31c compares the normalized execution time for different schemes, with results normalized over `Baseline`.

From the figure, `IR` exhibits 4% performance slowdown because it uses $Y = 3$ to avoid a large increase of background evictions, but smaller Y causes more reshuffles than `Baseline`, which has $Y = 4$. It has a negligible impact on space demand as it only shrinks the middle levels of the ORAM tree. `DR` lowers the space demand to 75% of `Baseline`, i.e., the state-of-the-art Ring ORAM implementation. `DR` achieves 25% space reduction, leading to a space utilization of 41.5%. We track the reclaimed blocks and observe that `DR` early reclaims most of the dead blocks in the system. `DR` is 3% slower than `Baseline`, with the overhead coming mainly from the enlarged buckets at the bottom levels. Reshuffling a bucket with the extended S value is more expensive than doing an original bucket. `NS` reduces the space demand of `Baseline` by 19% with comparable execution time. `NS` increases the number of *earlyReshuffles* but reduces the cost of *evictPath*. When combining `DR` and `NS`, AB-ORAM achieves 36% space reduction over `Baseline` while having around 4% performance overhead. The combined scheme achieves further space savings over `DR` and `NS`, indicating `DR` and `NS` are designs that improve space utilization from different directions. `AB` improves the space utilization of `Baseline` from 31.2% to 48.5% which is very close to 50%.

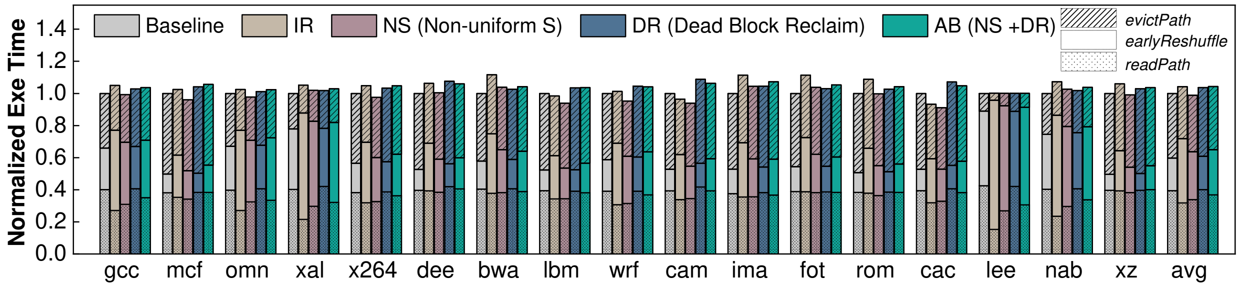
Note that the bucket size reduction in IR-ORAM originally was proposed upon Path ORAM, and in that setting, it benefits the performance. Because bucket size reduction is more significant in Path ORAM as it has a much smaller bucket size (Z=4). In addition, `IR` in the presence of the bucket compaction optimization in Ring ORAM incurs more dummy accesses due to stash overflow.



(a) Space reduction.



(b) Space utilization.



(c) Performance overhead with the breakdown of operations.

Figure 31: Space saving and performance overhead comparison of different schemes.

Figure 32 reports the bandwidth impact of our approach. From the figure, the extra bandwidth demand is negligible. On average, `AB` increases the bandwidth usage by 1%.

4.7.2 Bucket Reshuffle Impact

Figure 33 compares the number of reshuffles across the different levels for different schemes. `DR` has the closest number of reshuffles compared to `Baseline` due to S extension. `NS` increases the number of reshuffles for [L22, L23] where the S value is reduced by 2. `AB` compared to `NS` has more reshuffles for L21 and fewer reshuffles for L22 and L23. This is because we use L3-S1 in `AB` that shrinks S by 1 for [L21, L23].

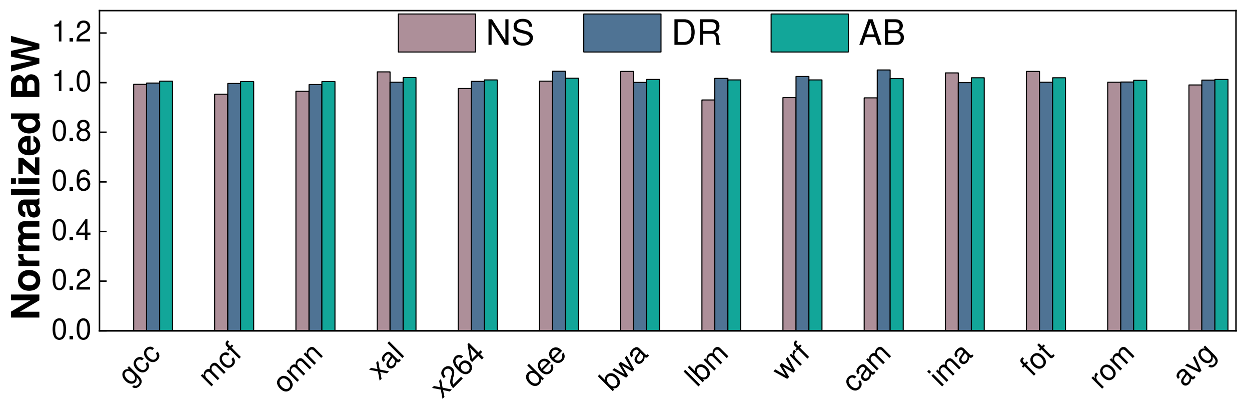


Figure 32: Bandwidth impact of AB-ORAM.

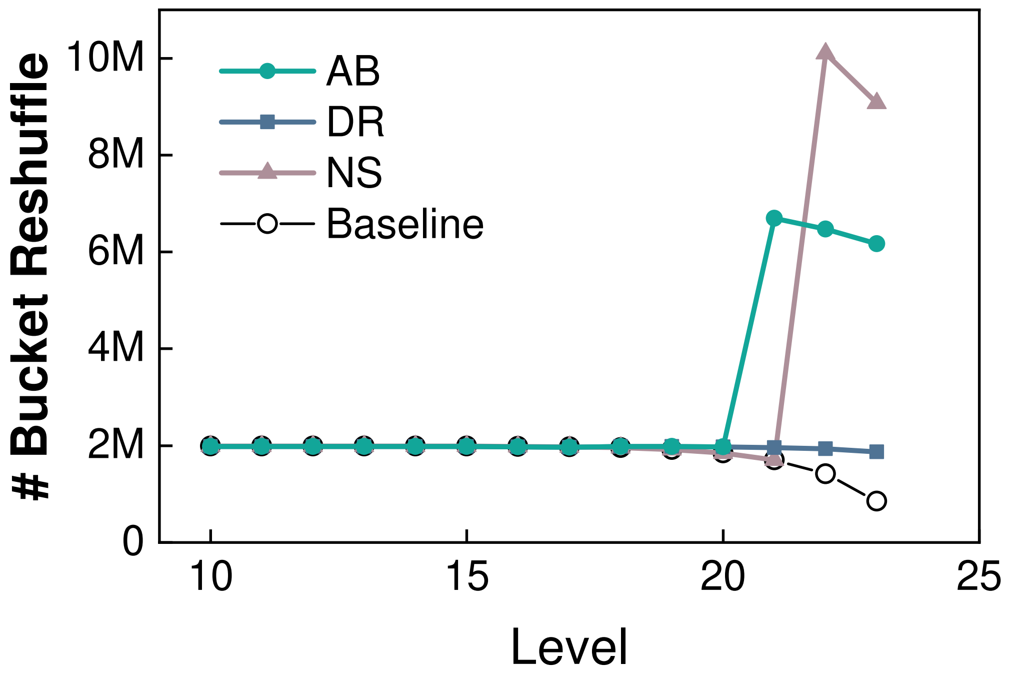
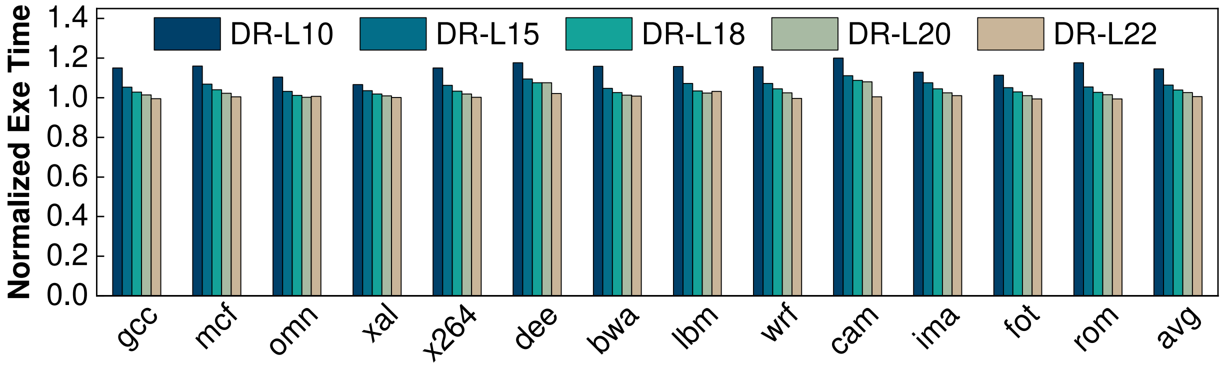


Figure 33: Comparing number of bucket reshuffles across the levels.

4.7.3 DR Sensitivity Analysis

Figure 34 denotes the result of a sensitivity analysis of `DR` scheme across the level choice. DR-L18 in Figure 34 is the same as `DR` in Figure 31. Top levels are less desirable for remote allocation due to their low contribution to space demand. For instance, the top 17 levels account for less than 1% of the space, while their reshuffle number contributes equally to performance as other levels.



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

Figure 34: Sensitivity analysis of DR to the number of levels.

4.7.4 Dead Block Lifetime Analysis

We studied the lifetime of dead blocks. A dead block's lifetime is defined as how long it has been invalid. In Figure 35 the X-axis indicates the tree levels while the Y-axis indicates the lifetime of dead blocks in terms of the number of online accesses. We report the minimum, average, and maximum lifetime of all dead blocks at each level. The three lines are the average of all benchmarks.

From the figure, the dead blocks from buckets above level 18 (i.e., closer to the root) 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 the leaves, the average lifetime is large, indicating that dead blocks at these levels tend to be *invalid* for a long duration.

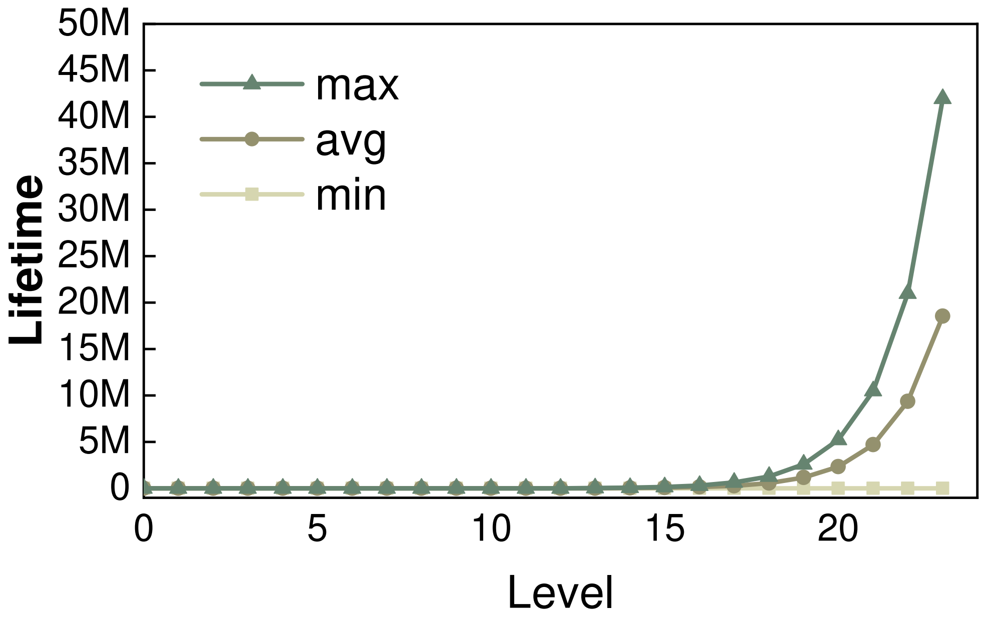


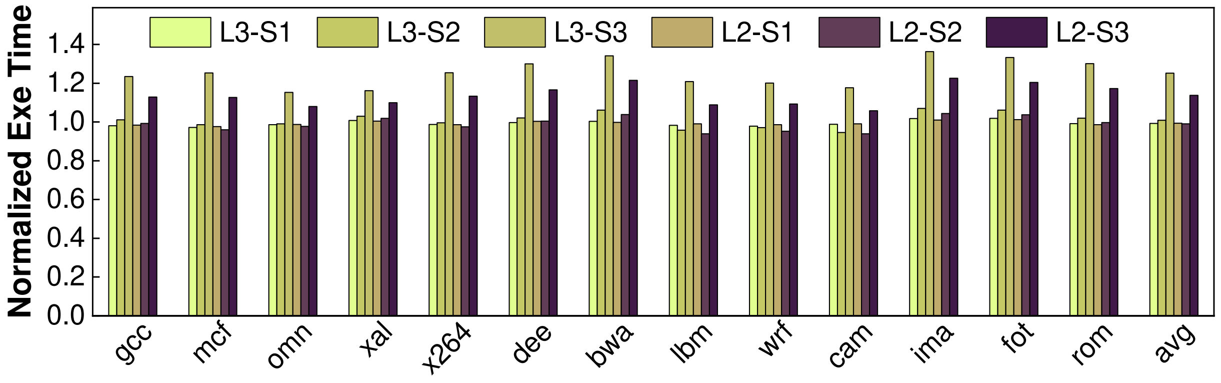
Figure 35: Dead blocks lifetime across ORAM tree levels.

4.7.5 NS Design Exploration

To determine the settings for `NS`, we studied different configurations and summarized the results in Figure 36. In this figure, `Ly-Sx` means that, for the last y levels, `NS` shrinks the S value by x. From the figure, an aggressive configuration, e.g., L3-S3, has large performance degradation. Note, it differs from Figure 27 because CB is the baseline here. We therefore chose L2-S2 for `NS` and L3-S1 for `DS+NS`, i.e., `AB`.

4.7.6 Remote Allocation Effectiveness

Figure 37 indicates the ratio of extended S values over the total number of bucket allocations. As shown in Figure 26, there are abundant dead blocks available at each level. Therefore, `DR` is able to extend almost all of the bucket allocations after gathering enough dead blocks in *DeadQ*. In contrast, when `NS` is also enabled, there are fewer dead blocks available at a time. Thus, `AB` has a lower extending ratio of 74%. Note, this ratio remains the



Normalized Space Demand					
L3-S1	L3-S2	L3-S3	L2-S1	L2-S2	L2-S3
59.4%	52.1%	44.8%	60.4%	54.2%	48.0%

Figure 36: Design exploration of NS.

same across different applications as the dead block availability is not application dependant.

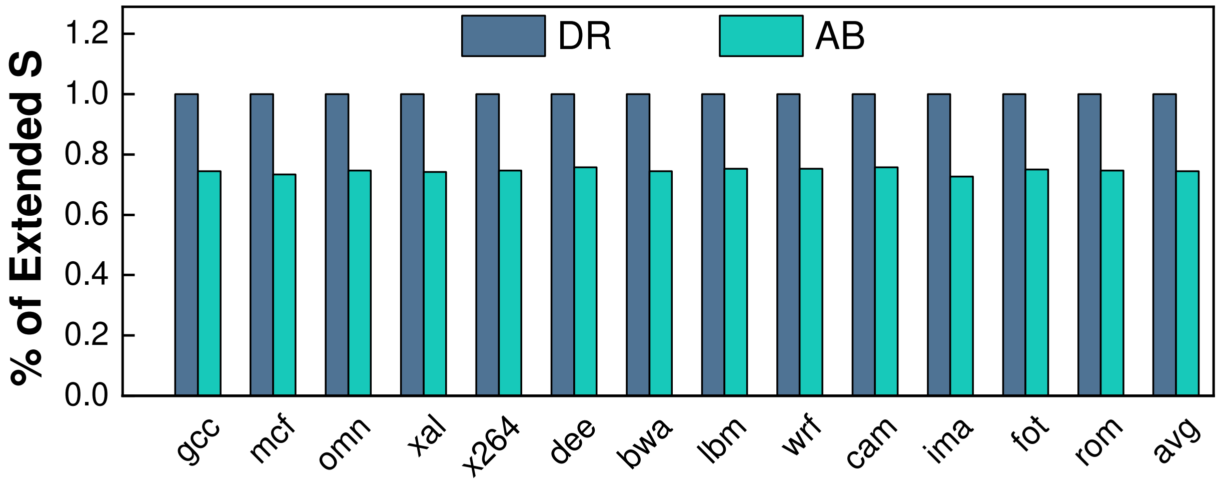


Figure 37: AB-ORAM capability for extending the S value.

4.7.7 Generalizability Over Different Applications

To assess the generalizability, we repeat the experiment with applications from another benchmark suite; PARSEC [6, 11]. Figure 38 reports the space and performance results. Our space saving remains the same as it is not application dependent. `NS` has a similar execution time as `Baseline`. `DR` and `AB`, on average, incur 3% and 4% performance overheads, respectively.

	NS	DR	AB
Space Reduction	19%	25%	36%
Space Utilization	38.5%	41.5%	48.5%

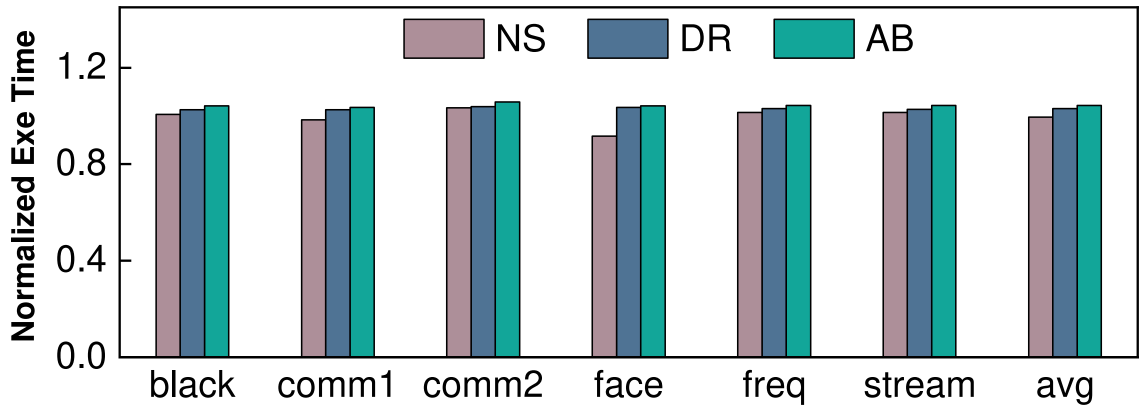


Figure 38: Generalizability analysis of AB-ORAM.

4.7.8 Storage Overhead

The remote allocation incurs two types of storage overhead – one is the *DeadQ* for tracking dead blocks, and the other is extra metadata for each bucket. The former is on-chip while the latter demands extra space in the main memory.

On-chip Overhead. Given we track dead blocks for 6 levels, and *DeadQs* are empirically set to have 1000 entries, the on-chip space is 21KB, which is small for modern processors.

Memory Overhead. For Ring ORAM, the bucket metadata takes 33B that fits into a block (i.e. 64B). 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 (33B + 28B) by setting $R = 6$ in Table 3.

4.7.9 Conclusion

In this chapter, we propose AB-ORAM to address the space inefficiency of Ring ORAM. 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 across the tree levels. It shrinks the buckets close to the leaves. AB-ORAM is orthogonal to the latest optimization of Ring ORAM and effectively lowers the overall space demand. On average, AB-ORAM achieves 36% space reduction over the state-of-the-art while introducing very low performance overhead.

5.0 EP-ORAM: Efficient NVM-Friendly Path Eviction for Ring ORAM in Hybrid Memory

5.1 DRAM Saving Overview

Among recent ORAM schemes [66, 58] that successfully reduced the protocol overhead to O(logN), where N is the number of protected data blocks, Ring ORAM [58] is a promising design. While its overall overhead remains O(logN), Ring ORAM achieves O(1) overhead for online accesses, i.e., the overhead to service user memory requests. Thus, Ring ORAM services user requests faster and improves the program performance, e.g., 2.7× improvement over Path ORAM [66]. The protocol maintenance operations are expensive but they are not always on the critical path.

However, Ring ORAM faces one major limitation, i.e., its low memory utilization. For a typical setting, the required DRAM space is 4.8× of the protected data [58]. Ring ORAM organizes the data blocks in a tree structure: each tree node, referred to as a bucket, consists of Z slots each of which can save one data block or dummy block. The low memory utilization comes from saving two types of dummy blocks. (1) When servicing a user memory request, Ring ORAM identifies the target tree path and fetches one block from each bucket along the path. Ring ORAM reserves S slots per bucket holding dummy data so that a bucket can service up to S accesses without reshuffling. (2) A data block, after its access, is randomly mapped to a different tree path. To prevent mapping to a tree path that has no empty slot, Ring ORAM doubles the number of memory slots that can hold user data. Therefore, on average, a tree bucket can hold $(Z\text{-}S)/2$ user data blocks. For a typical setting Z=12, S=7, the DRAM memory requirement of Ring ORAM is 4.8× of the protected user data.

To alleviate the low memory utilization in Ring ORAM, Cao *et al.* [7] proposed to shrink the bucket size such that the S dummy slots overlap with the slots that can save user data, which reduces the DRAM space by 34%. AB-ORAM proposed to exploit the dead blocks in the ORAM tree, which reduces the DRAM space by 22% [54]. Unfortunately, even with these designs, the DRAM space requirement remains high.

With the fast advances of NVM (non-volatile memory) technologies, e.g., ReRAM (Resistive Memory) [76] and STT-RAM (Spin-Transfer Torque Memory) [13], an alternative strategy to address the high DRAM space demand is to use NVM. In particular, for a DRAM/NVM hybrid memory system, if allocating the last two levels of the ORAM tree in NVM and the rest of the levels in DRAM, the required DRAM space can be reduced by 75%. However, NVM often suffers from large memory access latency, allocating too many levels, e.g., six levels, in NVM [32] can lead to 70% performance degradation. Given the performance of Ring ORAM is already 2.2× slower than the no-ORAM implementation, the large performance degradation makes an aggressive hybrid design less appealing.

In this chapter, we study the trade-offs in DRAM/NVM hybrid design as well as among the Ring ORAM operations. By exploiting the design space exposed by these trade-offs, we propose EP-ORAM, an efficient and NVM-friendly path eviction scheme to mitigate the high DRAM demand in Ring ORAM. In the following, we summarize our contributions.

- We study the interactions among different Ring ORAM operations. In particular, EvictPath (the operation to reshuffle tree paths) determines the stash size, the frequency of bucket level reshuffles, and the number of memory writes. We further identify the under-utilized middle levels of Ring ORAM. It exposes the opportunity to reduce the overhead of maintenance operations in Ring ORAM, in particular, the stash size.
- We propose EP-ORAM to exploit the design space exposed by our studies. EP-ORAM carefully partitions the ORAM tree between DRAM and NVM and shortens the path length during EvictPath operation, which not only reduces the number of NVM writes but also speeds up the operation.
- We evaluate the proposed design. Our experimental results show that, under the design constraints of no security compromise and similar performance as the baseline that saves two bottom levels in NVM, EP-ORAM helps to save three levels in NVM, achieving 50% DRAM space reduction. In addition, EP-ORAM reduces the NVM writes by 15%.

5.2 EP-ORAM

In this section, we first discuss our key observations and the design space challenges. We then elaborate on the EP-ORAM scheme to address them.

5.2.1 Trade-offs of Adopting ORAM in DRAM/NVM Memory

As explained in Section 2.3.3, the ORAM tree is a full binary tree so its capacity grows exponentially. In this tree, the capacity of the last level is equal to the capacity of all the prior levels. In other words, if we save the last two levels of the ORAM tree in NVM, we can save 75% of DRAM space. NVM often suffers from large memory access latency, allocating too many levels, e.g., six levels, in NVM [32] can lead to 70% performance degradation. Given the performance of Ring ORAM is already 2.2× slower than the no-ORAM implementation, the large performance degradation makes an aggressive hybrid design less appealing.

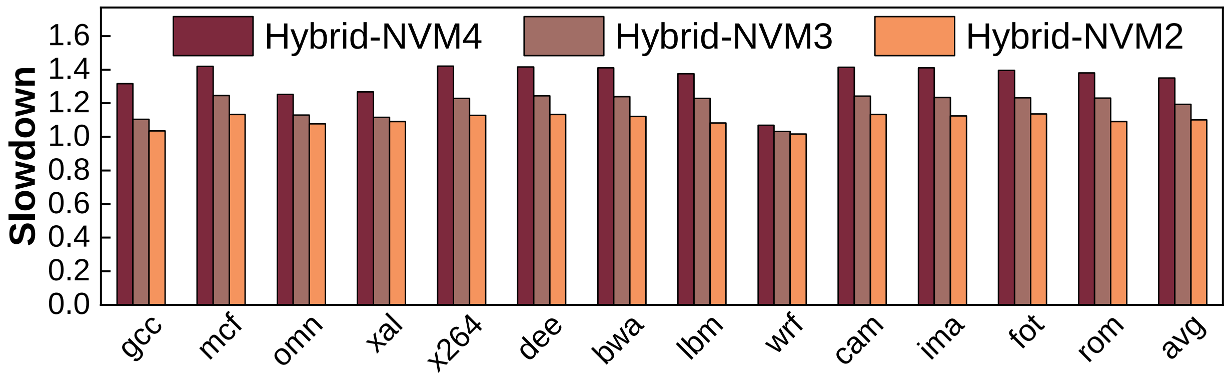


Figure 39: Slowdown of Ring ORAM in hybrid memory compared to DRAM (Hybrid-NVMx indicates saving x bottom levels in NVM).

Assume upgrading a 4-channel DRAM to a DRAM/NVM hybrid system, we reserve one memory channel for NVM traffic. Given NVM accesses are slower than DRAM ones, we expect to experience a modest performance slowdown in such system. Figure 39 shows the slowdown of ORAM on different hybrid configurations over the DRAM-only baseline. On average, Hybrid-NVM4 incurs 1.35× slowdown. Whereas, Hybrid-NVM3 and Hybrid-NVM2, on average, incur 1.19× and 1.10× slowdown respectively. If we bound the tolerable performance degradation to 10%, Hybrid-NVM2 is the only configuration that has an acceptable performance.

Another obstacle to adopting ORAM in a hybrid setting is the NVM writes imposed by the ORAM protocol. In a non-secure baseline each user memory request incurs at most one NVM write access. However, with ORAM, many more NVM writes are incurred per user program memory request. We studied the average number of NVM writes per user memory request for ORAM on different hybrid settings. Our experimental results showed that Hybrid-NVM4, Hybrid-NVM3, and Hybrid-NVM2, on average, incur 17.7×, 13.3×, and 8.8× NVM writes per user request, respectively. While NVM write endurance has improved significantly in recent years, for example, ReRAM has 10^9 write endurance, i.e., 10× better than that of PCM [77]. However, more than one order of magnitude NVM write increase could still be a big concern for the hybrid memory system.

In summary, modest performance degradation and modest NVM writes may be tolerable for a hybrid memory system. As such, we set 10% performance degradation and 10× NVM write increase as the design constraints. From the above discussion, We choose Hybrid-NVM2 as the baseline for comparison, i.e., saving the last two levels of the ORAM tree in NVM.

5.2.2 Trade-offs among Ring ORAM Operations

The three types of Ring ORAM maintenance operations help to tune the protocol to run smoothly. In particular, *EvictPath* flushes the blocks in the stash and resets the counters along the path. The frequency of *EvictPath* plays a key role in the design.

If we execute *EvictPath* less frequently, we expect to see more blocks accumulate in the stash and the counters of more buckets reach S, which triggers more *EarlyReshuffle* operations. However, *EarlyReshuffle* also flushes blocks from stash, though less effectively. If we execute *EarlyReshuffle* more frequently, we accordingly have more opportunities to flush the blocks in the stash.

BackgroundEvict serves as the last mechanism to ensure protocol correctness though it tends to introduce larger overhead. While we target minimizing the number of *BackgroundEvict*, the existence of *BackgroundEvict* operation exposes a large design space that we can explore across different Ring ORAM maintenance operations.

5.2.3 Under-utilized Middle Levels in Ring ORAM Tree

As mentioned in Section 2.3.3, on average, half of Z' entries in one bucket can hold real data, and the rest is filled by dummy blocks to ensure correctness. However, for one bucket at a given time, it may contain zero to Z' real block.

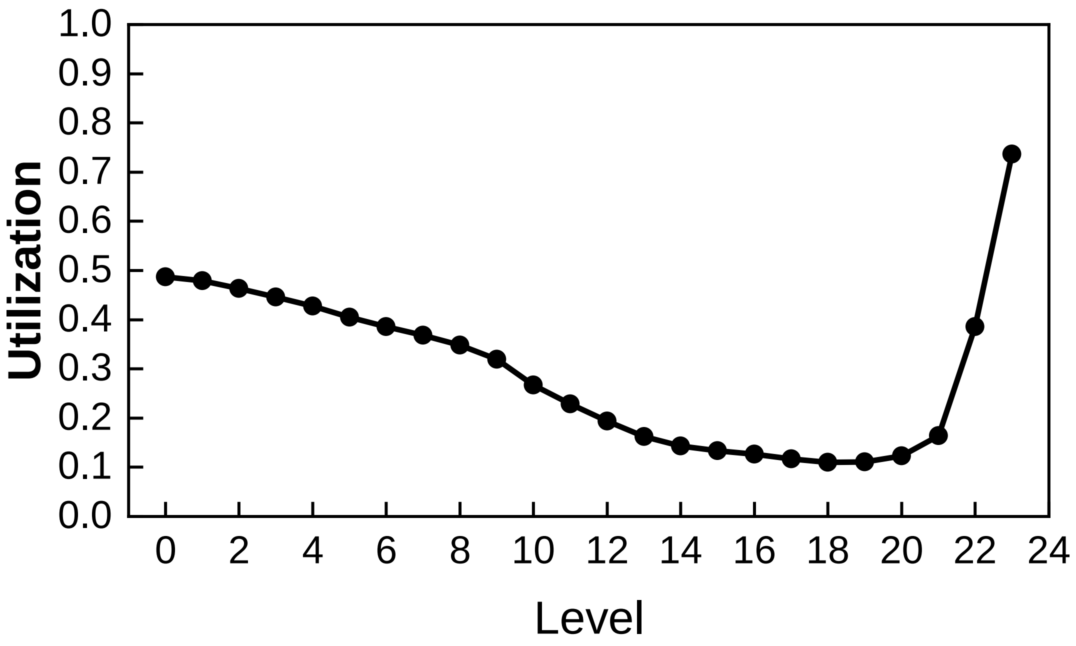


Figure 40: Bucket utilization across Ring ORAM tree levels (Utilization=1.0 means saving Z' blocks in one bucket).

We conducted an experiment to measure the average bucket utilization across different tree levels and reported the results in Figure 40. From the figure, we found that the bottom levels are highly utilized because they constitute the majority of the capacity. The top levels also have high utilization because they have high concentration of path overlaps. For example, a block in the stash can be written back to the root regardless of its path mapping (because all paths overlap at the root). As we move from the root to the middle levels, the utilization drops. Saving more blocks in these levels would be beneficial.

5.2.4 The EP-ORAM Design

To exploit the observations that we made in the preceding sections, we propose EP-ORAM, as shown in Figure 41. EP-ORAM consists of two key parameters (k, h), where k indicates the number of bottom levels to be saved in NVM; and h indicates the path length that *EvictPath* is to reshuffle. (0, 24) is the all-DRAM Ring ORAM implementation while (2, 24) is the baseline that saves the two bottom levels of the ORAM tree in NVM. Here, the ORAM tree has 24 levels, indicating it has 12 GB memory space, i.e., it protects 2.5 GB of user data.

By choosing $h < 24$, *EvictPath* exhibits two types of path reshuffles. They differ by the number of buckets to be read, re-encrypted, and written back.

- **Full Path:** This is the default *EvictPath* operation in Ring ORAM. It reads all buckets along one tree path, i.e., $L \times Z'$ blocks, flushes the stash blocks if possible, and re-encrypts and writes $L \times Z$ back to the path. All bucket counters along the path are reset after this operation.
- **Short Path:** This is similar to full path but only applies to the top h levels. That is, the buckets in top h levels are read, re-encrypted, and written back. The last $L - h$ levels are left untouched.

The k and h values. For a DRAM/NVM hybrid system, k and h are two independent meta parameters. That is, it is possible to partially reshuffle a short path that consists of only DRAM levels, or be extended to the NVM level. On the one hand, by having the short path consisting of only DRAM levels, the partial reshuffle is fast and incurs zero NVM writes. However, for a DRAM/NVM hybrid system that has an independent memory channel for NVM, the NVM memory channel stays idle during the long *EvictPath*, which wastes the memory parallelism that can be potentially exploited to improve the effectiveness of the design. On the other hand, extending the short path to the NVM level incurs more NVM writes. It could become a major concern if the number of NVM write is too high.

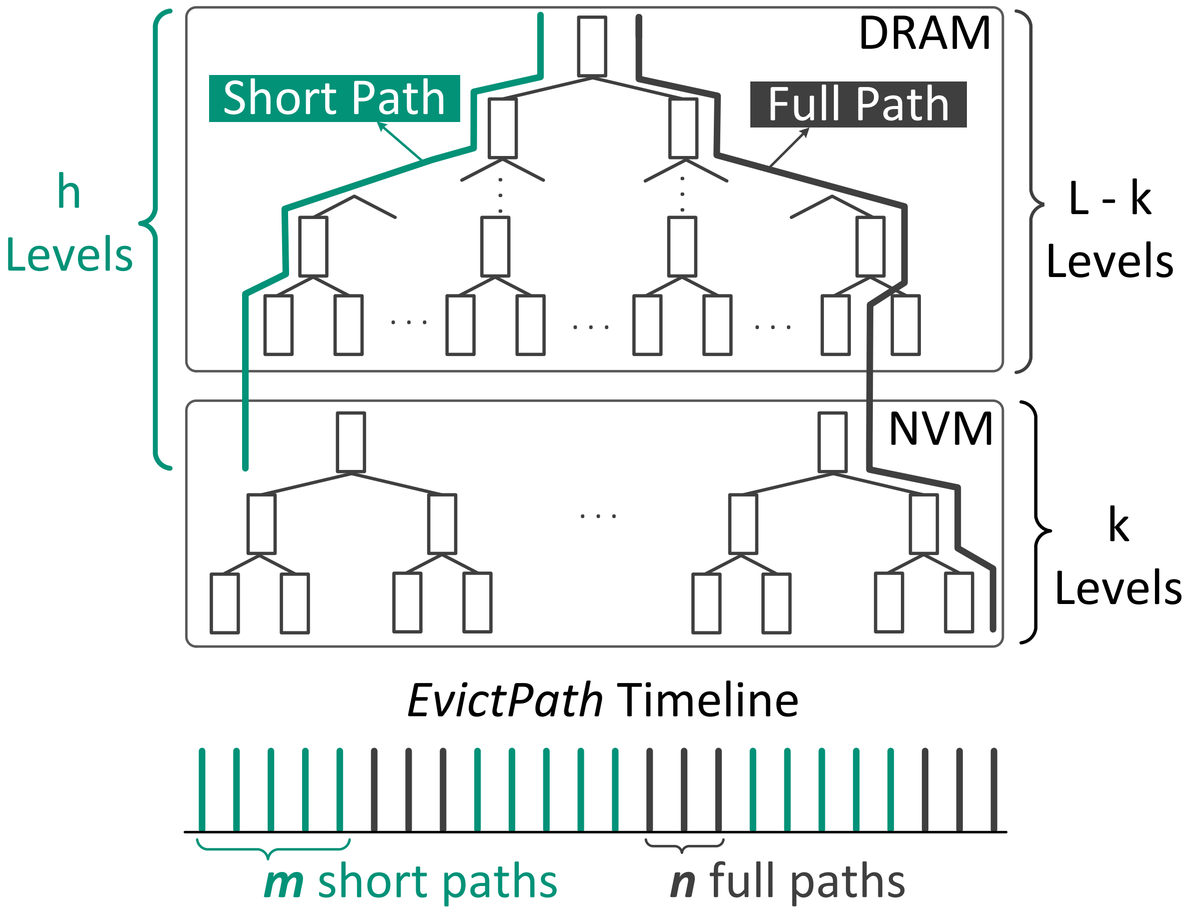


Figure 41: An overview of the EP-ORAM design in hybrid memory system.

Short/Full path pattern. There are two direct impacts of adopting short path *EvictPath*: (1) The buckets from bottom $L - h$ levels are left untouched and thus their counters are not reset. There is an increasing possibility of triggering *EarlyReshuffle* operations. (2) The blocks in the stash that may be flushed to the bottom levels can now be flushed to $L - h$ and above levels. This effectively increases the utilization of these levels. Due to their limited sizes, there is a possibility the blocks may not be flushed and thus stay in the stash longer. An overflow of the stash may trigger *BackgroundEvict* and degrades the overall performance. For this purpose, we propose to conduct m short path *EvictPath* and then n full path *EvictPath* and repeat continuously, as shown in Figure 41. Note, there are five *ReadPath* operations between every two *EvictPath* invocations, the same as the baseline Ring ORAM.

To ensure the same security protection as that of the baseline Ring ORAM implementation, the invocation of short path *EvictPath* cannot be on-demand. That is, we need to statically decide a fixed pattern and apply it to all benchmarks.

Intuitively, the bigger the m value is, the more we reduce the number of NVM accesses. However, a big m value leads to a higher chance of increased space utilization of middle levels and possible stash overflow. We will study these parameters in the experiment section.

5.2.5 Security Analysis

In this section, we discuss how EP-ORAM preserves the same security guarantee as Ring ORAM. First we show the partitioning of the ORAM tree in DRAM/NVM is secure. Then, we show different path length *EvictPath* scheduling is secure.

In all tree-based ORAMs, the security guarantee is that path accesses are indistinguishable so that an attacker cannot infer any information about the user program requests. EP-ORAM does not affect the online access i.e. *ReadPath* operation so these path accesses remain indistinguishable the same way they were in Ring ORAM.

Regarding *EvictPath* operation, path lengths of evicted paths are changed in EP-ORAM but they follow a fixed pattern. Note that in Ring ORAM *EvictPath* operations are statically scheduled in a reverse-lexicographic order. Therefore, what path is going to be evicted next is public information. In the same manner, in EP-ORAM, it is public information what paths are going to be accessed in full or short length. This pattern is fixed during the execution and remains the same for all programs. Therefore, EP-ORAM does not leak any extra information to the attacker.

EarlyReshuffle activation is also public knowledge. Any bucket that is accessed S times is going to be reshuffled. EP-ORAM does not change *EarlyReshuffle* operation and thus leaks no extra information.

Naturally with the reduced number of *EvictPath* on bucket of bottom levels the number of *EarlyReshuffle* they get may increase. Nevertheless, this does not disclose any information regarding the user program since the number that a bucket is reshuffles is already known by public in Ring ORAM baseline.

While EP-ORAM potentially increases the utilization in the middle tree levels, the change of the overall utilization is known but the utilization for a given bucket, i.e., how many real data blocks saved in this bucket, remains unknown to the attackers. From what an attacker can observe from the outside, all blocks in a 12-entry bucket remains indistinguishable.

The invocations of the *BackgroundEvict* operation do not incur any security issues because, similar to the Ring ORAM baseline, *BackgroundEvict* is indistinguishable from normal accesses from the perspective of an outside observer. This indistinguishability persists in EP-ORAM as well; EP-ORAM does not alter how this operation is conducted. The only difference between the baseline and EP-ORAM is that EP-ORAM may experience more *BackgroundEvict* invocations. While this increase can have adverse effects on performance and marginally diminish the performance improvements, it does not compromise security. This is because, like in the baseline, the *BackgroundEvict* operation remains indistinguishable from normal user access. Therefore, the potential increase in the number of *BackgroundEvict* operations does not disclose any information.

In summary, EP-ORAM preserves the security guarantee of Ring ORAM.

5.3 Evaluation

We used trace-based simulation to evaluate EP-ORAM, similar to those in the literature [72, 7]. We used the Pin tool [15] for collecting traces from SPEC CPU2017 [1]. For each benchmark, we gathered 40 million memory access traces after skipping the warm-up phase. We ran each trace repeatedly to form a 400 million trace to place the ORAM tree into a stable state. We fed the last million accesses to USIMM [11] for DRAM access simulation. We modeled a 4-issue OoO (out-of-order) 3.2GHz processor with 160 ROB entries; a 4-channel memory with one channel is dedicated to NVM. We adopted ReRAM as the NVM and added a fixed latency of tWR = 200ns derived from [77] to simulate NVM accesses in USIMM. Table 7 lists the rest of the system configurations.

Table 7: System configuration for EP-ORAM evaluation.

ORAM Configuration		Processor Configuration	
ORAM tree	24 levels	L1 D-cache	2-way 256 KB
Tree top cache	10 levels	L2 (LLC)	8-way 2 MB
Block size	64B	Mem channels	3 DRAM, 1 NVM
Stash entries	300	NVM	ReRAM (10^9 writes)

Following the typical setting [58, 54], we modeled a 24-level Ring ORAM tree with Z=12, and Z'=5, S=7. Given that each block is a 64B, the total ORAM tree size is $(2^{24} - 1) \times 12 \times 64$B = 12 GB. Only 50% of all Z' entries contain user data. Thus, the protected user space is 2.5 GB. We cached the top 10 levels of the tree on-chip. We evaluated the following schemes.

- **All-DRAM:** it implements Ring ORAM with all tree levels being stored in DRAM.
- **Hybrid-NVM2:** it implements Ring ORAM with 2 levels in NVM and the rest of the levels in DRAM.
- **EP-ORAM:** it implements Ring ORAM and adopts EP-ORAM with k=3, and h=21. The short/full path pattern is that it uses $m = \infty$, and n=0.

5.3.1 DRAM Space, NVM Traffic and Performance Analysis

Figure 42 compares the DRAM space demand for different schemes. From the figure, EP-ORAM reduces DRAM demand to only 1.5 GB, exhibiting 50% and 87.5% reductions over Hybrid-NVM2 and All-DRAM, respectively. This greatly alleviates the DRAM space demand in Ring ORAM designs.

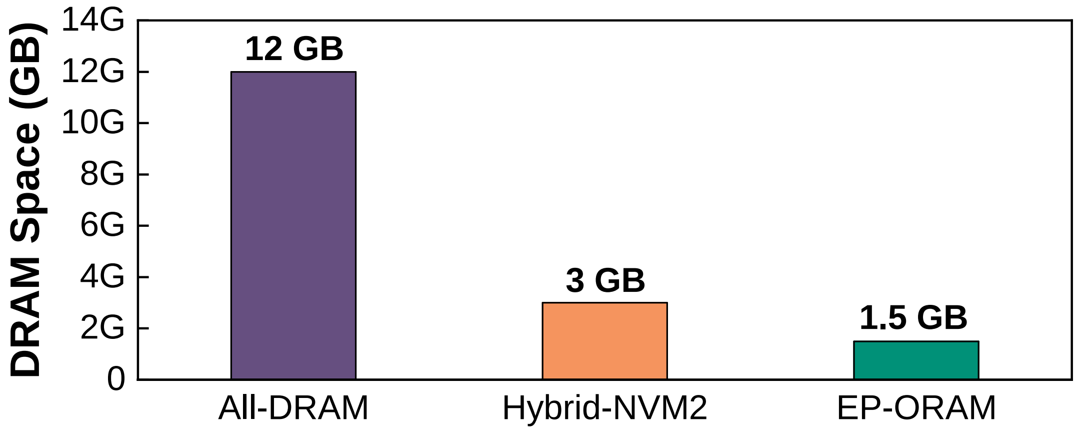


Figure 42: DRAM space demand of different schemes.

EP-ORAM achieves large DRAM savings under the design constraints in Section 5.2.1, i.e., 10% performance slowdown over All-DRAM, and 10× user memory requests. Figure 43 reports the number of NVM writes in EP-ORAM with the result being normalized over Hybrid-NVM2. On average, EP-ORAM reduces the NVM write traffic by 15%. This reduction comes mainly from that short path *EvictPaths* generate no NVM writes. However, the number of *EarlyReshuffles* for buckets in NVM increases as bucket counters are not frequently reset. The reduction over-weighs the increase when k=3. The number of NVM writes in EP-ORAM is 7.7× user memory requests. Figure 44 reports the slowdown of Hybrid-NVM2 and EP-ORAM over All-DRAM baseline. On average, Hybrid-NVM2 and EP-ORAM incur 10% and 8% slowdowns over All-DRAM.

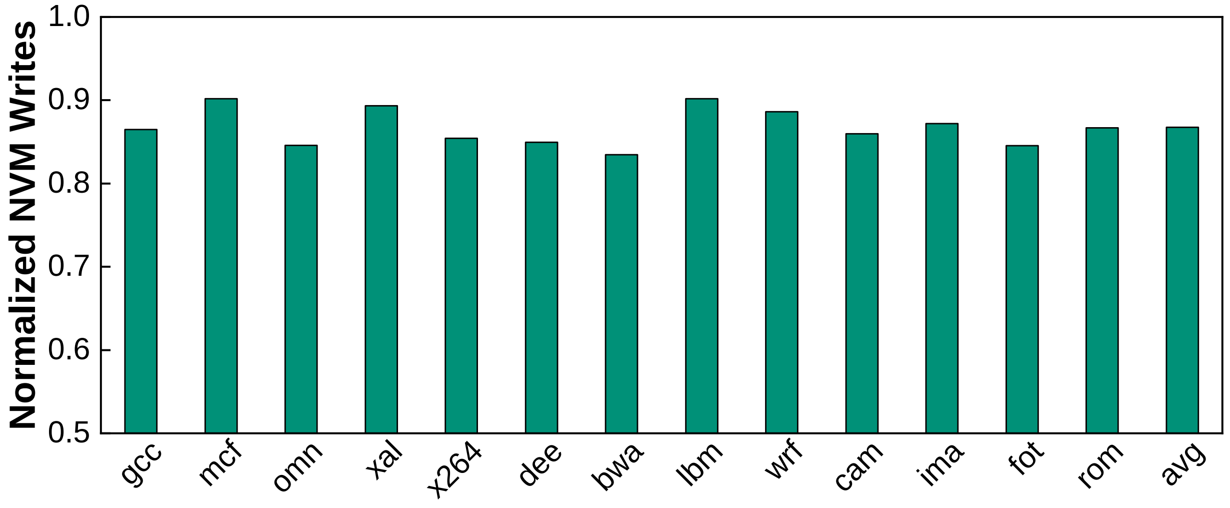


Figure 43: NVM writes reduction of EP-ORAM compared to Hybrid-NVM2.

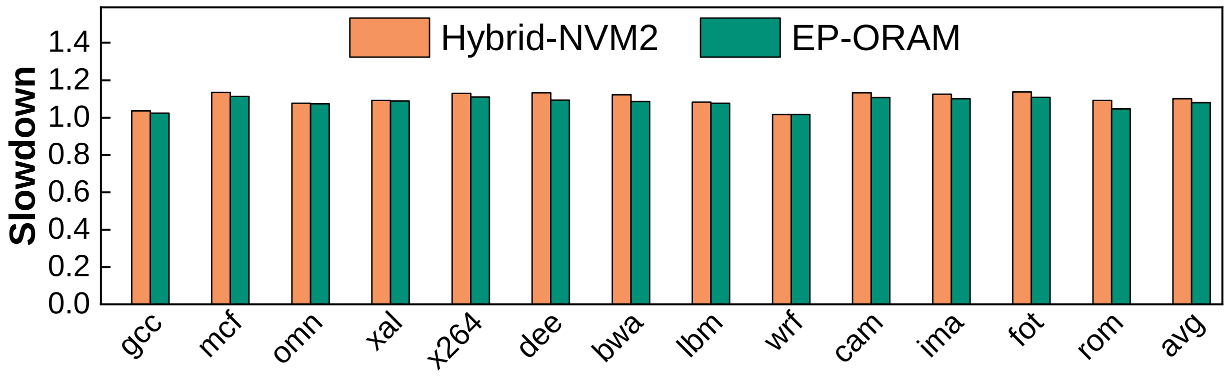


Figure 44: Performance comparison of EP-ORAM and Hybrid-NVM2.

5.3.2 EP-ORAM Design Exploration

In this section, we study how different design choices affect EP-ORAM. Figure 45 compares the number of NVM writes using different m and n values, with the result being normalized over Hybrid-NVM3 (i.e., saving the bottom 3 levels in NVM, and no EP-ORAM). The shaded portion of each bar indicates the amount of writes from *EvictPath* while the rest comes from *EarlyReshuffle*. For each configuration $mAnB$ indicates m=A, and n=B. From the figure, with increasing m values, the number of NVM writes from *EvictPath* decreases whereas that from *EarlyReshuffle* increases. On average, the total number of NVM writes decreases as we enlarge m value. In particular, $m\infty n0$ reduces the NVM writes to 58% of those in Hybrid-NVM3. Figure 46 compares the execution time of different configurations of EP-ORAM with the result being normalized over Hybrid-NVM3. From the figure, $m\infty n0$ performs the best and reduces the execution time by 10%.

However, we observed that `lbm` has more NVM writes and larger slowdown at $m\infty n0$ over those at $m50n1$. This is due to more invocations of expensive *BackgroundEvict* operations.

The k, h parameters in EP-ORAM design are independent. Figure 47 compares the following configurations: Hybrid-NVM4 and EP-ORAM with k=4, h=21. Hybrid-NVM4 incurs 35% slowdown compared to All-DRAM. EP-ORAM reduces this degradation to 11% on average. While it is slightly over our design constraint (10% performance degradation), it reduces the DRAM space by 93.75% over All-DRAM, or 768MB rather than 12GB in All-DRAM.

5.3.3 Utilization Analysis

To study the impact of EP-ORAM on bucket utilization, we repeated the experiment in Figure 40 with EP-ORAM. Figure 48 summarizes the results. From the figure, EP-ORAM makes better use of middle levels. There is a spike at level 20 because Ring ORAM aggressively writes blocks to the bottom level and levels 23 and 20 are the bottom levels for the full path and short path *EvictPath*, respectively.

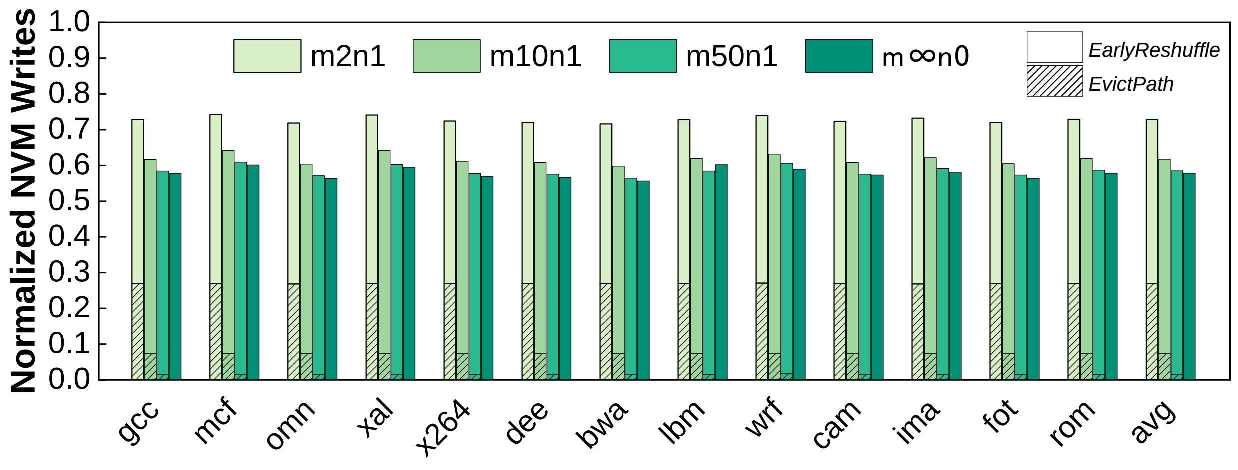


Figure 45: NVM writes reduction with different configurations of EP-ORAM compared to Hybrid-NVM3.

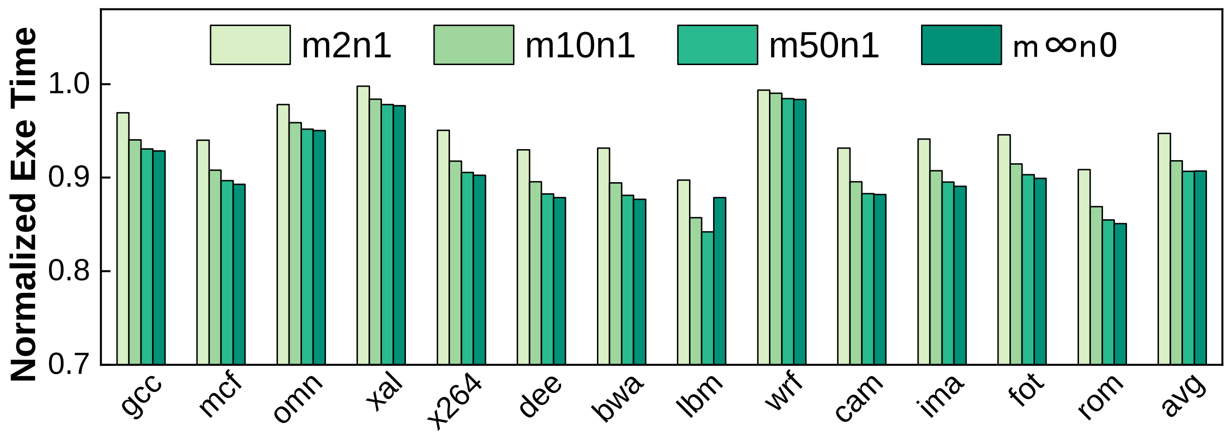


Figure 46: Performance improvement with different configurations of EP-ORAM compared to Hybrid-NVM3.

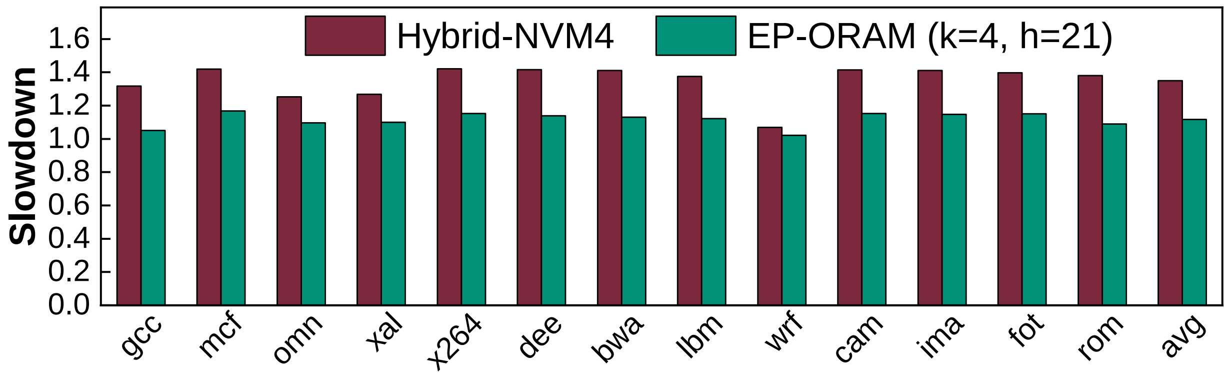


Figure 47: Comparing EP-ORAM and Hybrid-NVM4 slowdown.

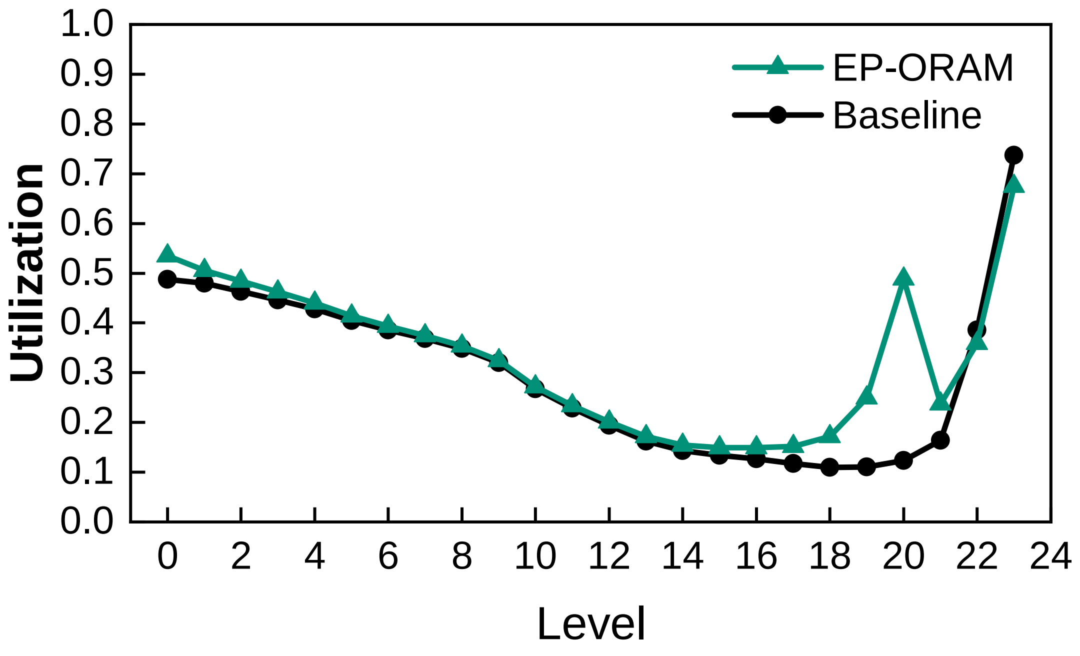


Figure 48: Bucket utilization of EP-ORAM and the baseline across levels.

5.3.4 Conclusion

In this chapter, we propose EP-ORAM [55] to enable efficient adoption of Ring ORAM in DRAM/NVM hybrid memory. EP-ORAM partitions the ORAM tree such that bottom levels are stored in NVM to save DRAM space. EP-ORAM identifies an opportunity in existing trade-offs among Ring ORAM operations to shorten the EvictPath operation. EP-ORAM achieves 50% DRAM space saving and reduces NVM writes by 15%.

6.0 Conclusions

6.1 Summary

In this dissertation, efforts were made to enhance the efficiency of secure memory with ORAM from a system perspective, aiming to increase its appeal for widespread adoption. It has been elaborated how encryption and authentication mechanisms alone are insufficient to fully safeguard user privacy, particularly against access pattern attacks. Given that ORAM serves as a robust method for concealing access patterns in scenarios where users delegate their data and computation to untrusted memory, the emphasis was placed on enhancing its efficiency in modern computers through the introduction of architectural techniques.

In this dissertation, I introduced three innovative approaches to optimize Oblivious RAM (ORAM) implementations, each targeting distinct challenges associated with memory intensity, space inefficiency, and hybrid memory adoption. First, IR-ORAM focuses on mitigating the high memory intensity common in Path ORAM through a set of techniques including IR-Alloc, IR-Stash, and IR-DWB, which collectively enhance performance by optimizing data block access, buffering strategies, and converting dummy path accesses. Second, AB-ORAM addresses the space inefficiency issue of Ring ORAM by reclaiming dead block space and adopting a non-uniform bucket size strategy, thereby reducing overall space demand with minimal performance impact. Finally, EP-ORAM facilitates the efficient adoption of Ring ORAM in hybrid DRAM/NVM memory systems by partitioning the ORAM tree to leverage NVM for bottom levels, saving DRAM space and reducing NVM writes. These approaches not only achieve significant improvements in their respective areas but also maintain the essential security protections and memory access obliviousness, demonstrating their effectiveness and compatibility with existing ORAM optimizations.

6.2 Future Work

6.2.1 ORAM for Recommendation Systems

The threat of attacks on user privacy can be a major obstacle to outsourcing deep learning-based recommendation systems to the cloud [84]. As discussed in 1.1, the access pattern of embedding tables in deep learning-based recommendation systems is another instance of an access pattern posing a threat to user privacy [31, 40, 53]. Given the prevalence and increasing ubiquity of recommendation systems, protecting the access pattern in such configurations is deemed necessary. However, incorporating ORAM for access pattern protection in recommendation systems introduces new challenges. This section elaborates on these challenges, as well as the potentials that can be exploited to develop an embedding table-tailored ORAM design.

Deep learning-based recommendation systems leverage neural network to provide users with personalized recommendations based on their past behavior and preferences. They have application in many domain such as online shopping, and video and music streaming platforms. Deep learning takes both continuous and categorical features as their inputs. Embedding tables are used to transform the categorical features to enhance their representation.

Embedding tables are a crucial component in deep learning-based recommendation systems. They are used to transform sparse categorical features, such as movie genres or user IDs, into dense vector representations [16]. This transformation is essential because it allows the model to capture the relationships between different categories. For instance, with a movie recommendation system, each movie is associated with one or more genres, such as Action, Comedy, Drama, Romance, and Thriller. In the context of embedding tables, each of these genres would be mapped to a dense vector in a high-dimensional space. During the training phase, the model learns the optimal embedding table entry for each category based on the training data. The goal is to adjust the embedding table entry in such a way that the model's predictions align with the observed user-item interactions in the training data. For instance, the system might learn that Action and Thriller are often liked by the same users, so their embedding table entries would be close together in the embedding space, meaning that the numerical content of their embedding table entries is similar, often measured using metrics such as Euclidean distance. During the inference phase, the model uses the learned embedding table to make predictions.

The size of the embedding table varies depending on the deep learning model. However, they can grow very large, which is inevitable because a large embedding table can lead to higher accuracy. Given that embedding tables can be very large, up to hundreds of gigabytes [30, 50, 48, 85, 17, 80, 29], it is unrealistic to assume that all applications can benefit from on-device machine learning inference. Therefore, it is natural for many applications to take advantage of cloud inference to leverage their capacity to hold large models with large embedding tables [40, 51]. With this comes the ever-growing concern of user privacy. As studies have shown, the access pattern of an embedding table can reveal sensitive information about the user. This is because the index of an embedding table entry discloses features associated with a user.

ORAM, by design, requires at least twice the memory size it intends to protect. This presents a significant challenge when implementing ORAM in settings with large embedding tables due to the increased space demand. Additionally, machine learning inference is typically sensitive to latency. A recommendation system that takes an excessive amount of time to provide recommendations would not be beneficial. Therefore, managing latency overhead becomes another critical challenge when using ORAM to protect the access patterns of embedding tables.

In the preliminary study of embedding tables, we observed a key difference from CPU benchmark traces. Specifically, accesses to embedding tables exhibit minimal locality. Furthermore, the sequence of accesses to the embedding table more closely resembles a random trace. Our study found implications due to this matter. Due to the lack of locality, the PLB hit rate is extremely low when embedding table traces are fed to the ORAM simulator. This implies that the PosMap accesses cannot be skipped for most user block accesses. In addition to the low PLB hit rate, the depth of the PosMap table lookup increases significantly with a large embedding table. This is because there is limited on-chip space to accommodate the last level of PosMap. Therefore, to cover the PosMap information of a large embedding table, the number of levels in the hierarchy increases. For instance, a 40 GB embedding table would require 4 levels of PosMap, with the last one being stored on-chip and occupying 32 KB. This means each memory access (out of the embedding table trace) will require 3 PosMap accesses followed by 1 data access, assuming there is no PLB.

We discovered that due to the random nature of embedding table traces, the PLB is not effective at all. Our study revealed that PosMap accesses constitute 64.3% of all accesses on average for embedding tables of the Taobao recommendation system [62]. If there is no PLB in place, PosMap accesses would constitute 66.6% of all accesses. The key observation here is that within the context of safeguarding embedding table accesses with ORAM, PosMap accesses serve as the primary source of overhead. Thus, reducing PosMap accesses should be prioritized, as they constitute the main source of overhead.

6.2.2 Merkle Tree for Position Map Access Reduction

In this section, an idea will be discussed to reduce PosMap accesses when protecting embedding tables' access pattern with ORAM in deep learning-based recommendation systems.

As discussed in Chapter 2, when the user data that we want to protect grows large, so does the size of the PosMap table. Since the PosMap blocks hold secret data, i.e., block-to-PathID mapping, it must also be protected with ORAM. Thus, the PosMap table is constructed in a recursive manner until the last level PosMap fits on-chip.

Let us illustrate how this works through an example. Assuming we want to protect 32GB of data with a block size of 128B. Then we will need to protect 256 million (32GB / 128B) data blocks. Therefore, the first level of the PosMap table needs to keep the record for 256 million block-to-PathID mappings. Given that the PathID takes 32 bits, i.e., 4B, each block can hold 32 (128B / 4B) numbers of block-to-PathID mappings. Assuming our on-chip budget is in the range of 32KB to 64KB, we will need 4 levels of PosMap tables. The following indicates the size calculation of each level:

- **PosMap-1:** 32GB/32 = 1GB
- **PosMap-2:** 1GB/32 = 32MB
- **PosMap-3:** 32MB/32 = 1MB

- **PosMap-4:** 1MB/32 = 32KB

PosMap-4 will be stored on-chip, while the remaining levels will be stored in the ORAM tree alongside the 32GB user data, as introduced in [21]. This means that without the presence of a PLB for caching the PosMap blocks, we will need three path accesses to retrieve the PathID of the block we want to access prior to the data path access, as demonstrated in Figure 49. This means without a PLB, there are 4 path accesses required per user data block request and 3 of them are PosMap accesses.

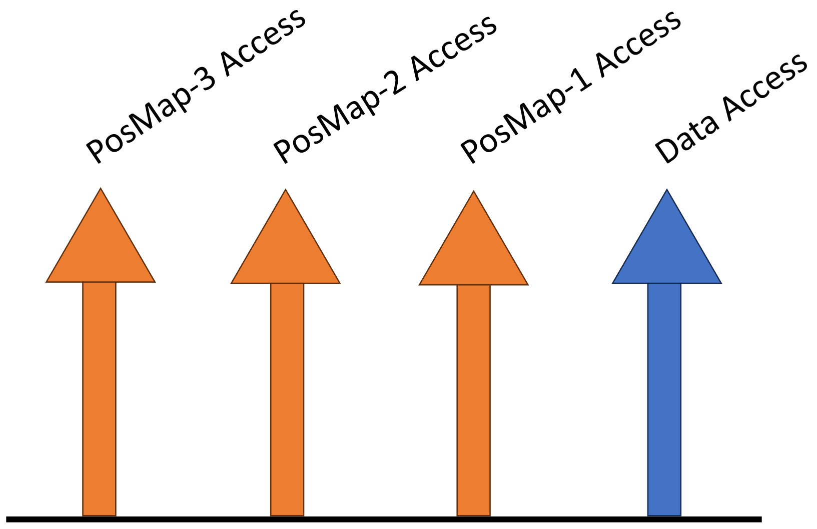


Figure 49: Timeline of path access in a large embedding setup with a 4-level PosMap table.

The idea is to redesign the PosMap structure in order to reduce the number of levels in the PosMap table, thereby reducing the number of accesses required per user data request. To achieve this reduction in levels within the PosMap, we must increase the number of block-to-PathID mappings stored per 128B block. To do so, we propose employing the Merkle Tree approach [60, 23] for constructing the PathIDs instead of storing them directly as 4B numbers in the PosMap block. Thus, instead of saving the PathID directly, we generate the PathID from a secure one-to-one mapping. Here is the formulation for the secure function:

$$PathID = secureFunc(Nonce, BlockIndex, BlockCtr)$$

In the function *secureFunc*, there are three inputs, each of which is outlined in the following discussion. Figure 50 depicts the PosMap block layout in this scheme, where each PosMap block consists of a 26-bit Nonce, similar to the concept of a major counter in the Merkle tree, and 128 numbers of 10-bit BlockCtr, akin to the minor counters in the Merkle tree implementation. Therefore, each PosMap block is able to retain the required information to construct the PathID of 128 blocks. Note that the input BlockIndex in *secureFunc* is implied through the location of the corresponding block's BlockCtr in the PosMap block, as illustrated in Figure 50.

All Nonces of all PosMap blocks are initialized with a random number, and all BlockCtrs are initialized to zero. Whenever there is an access to a data block, its PathID is first calculated through *secureFunc*, and then the corresponding BlockCtr is incremented. Once any of the 128 BlockCtrs overflows, the Nonce should be incremented, and all BlockCtrs in the PosMap block must be reset. Therefore, in that case, there will be an overhead of updating the blocks' PathID over the ORAM tree. However, given that in embedding tables the access pattern is quite random, one could expect that the counter overflow is very rare. Thus, the overall maintenance overhead of dealing with the overflow is extremely low.

The following discussion details the implementation of *secureFunc*. The first step is to concatenate the two inputs: BlockIndex, and BlockCtr, and then XOR the result with Nonce. However, it would not be secure to use the raw concatenation as the final PathID, simply because then the consecutive PathIDs of a block would appear incremental. Note that each BlockCtr is incremented after each PosMap access, and that is how a block is remapped to a new PathID. Thus, in order to ensure the remapping is secure and consecutive PathIDs appear non-incremental and rather random, we apply a secret encryption mapping to the generated PathID from the concatenation. This encryption function is kept secret and resides in the trusted base.

One minor detail is that the proposed layout for the PosMap block is slightly larger than 128B. However, this issue can be handled through block compression at the CPU side in the trusted base. Compression is feasible, given that the majority of BlockCtrs in a given

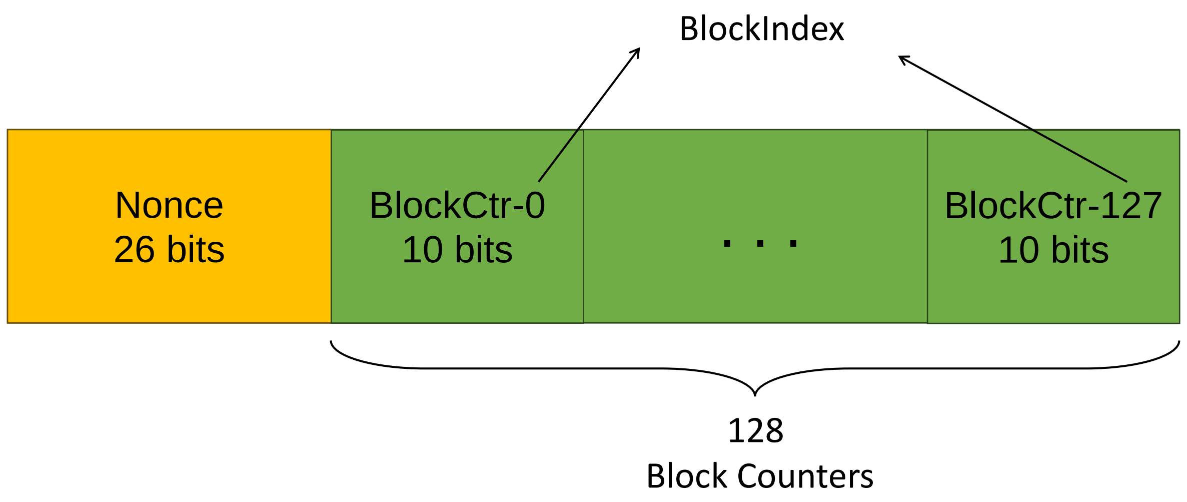


Figure 50: The PosMap block layout with the Merkle tree PosMap scheme.

PosMap block are expected to have many zero bits due to the random access pattern of embedding tables, i.e., the lack of locality.

Let us revisit the discussion regarding the calculation of PosMap levels for the proposed scheme. Previously, we needed 4 levels of PosMap accesses. With this scheme, we now need one less level because each PosMap block can hold the PathID of 128 blocks instead of 32. Here is the PosMap size calculation for this scheme:

- **PosMap-1:** 32GB/128 = 256MB
- **PosMap-2:** 256MB/128 = 2MB
- **PosMap-3:** 2MB/128 = 16KB

With the proposed scheme, the PosMap access overhead will be reduced due to the fewer number of PosMap levels. Previously, one user request could result in 4 path accesses according to Figure 49, in the worst-case scenario where all PosMap accesses are PLB misses. In contrast, with the Merkle PosMap scheme, the worst-case scenario will result in 3 path accesses. This represents a 25% reduction in the number of path accesses. Figure 51 illustrates the preliminary result. From the figure, it can be observed that the total number of memory accesses with the proposed scheme is reduced by 29% compared to the baseline.

The result indicates that the reduction in memory access overhead is 4% higher than the expected potential (i.e. 25% reduction). Here is why: with the proposed scheme, not only does the number of path accesses reduce in the worst-case scenario, but also the occurrence of worst-case scenarios decreases because of an increase in PLB hit rate. That is expected because with the proposed scheme, each PosMap block covers PathID for a wider range of space. Thus, it is only expected that we observe a higher PLB hit rate.

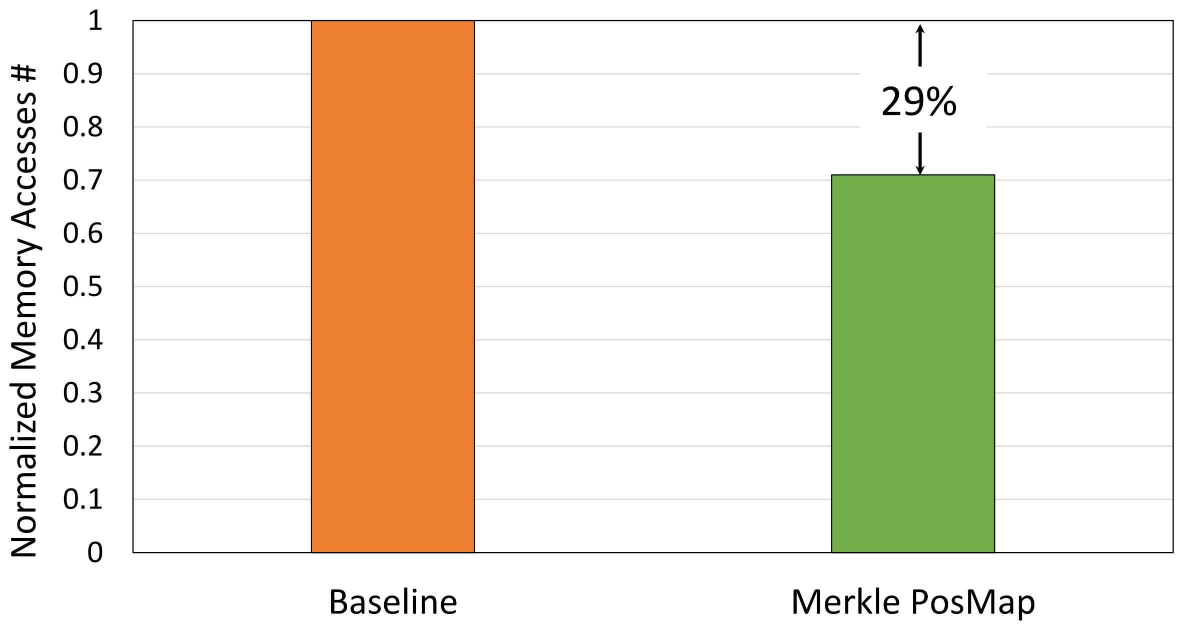


Figure 51: The total memory access reduction achieved with the Merkle PosMap scheme.

In this section, I introduced a Merkle tree-based PosMap construction scheme to reduce the number of PosMap accesses for protecting large embedding tables in deep learning-based recommendation systems with ORAM. The preliminary results have proven to be promising. Therefore, there is potential for future work to follow this approach.

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] Kholoud Saad Al-Saleh and Abdelfettah Belghith. Radix path: A reduced bucket size oram for secure cloud storage. *IEEE Access*, 2019.

[4] 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.

[5] 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.

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

[7] 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.

[8] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 668–679, New York, NY, USA, 2015. Association for Computing Machinery.

[9] Anrin Chakraborti, Adam J. Aviv, Seung Geol Choi, Travis Mayberry, Daniel S. Roche, and Radu Sion. roram: Efficient range oram with o(log2 n) locality. *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[10] Anrin Chakraborti and Radu Sion. Concuroram: High-throughput stateless parallel multi-client oram. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[11] Niladrish Chatterjee, Rajeev Balasubramonian, Manjunath Shevgoor, Seth H. Pugsley, Aniruddha N. Udipi, Ali Shafiee, Kshitij Sudan, Manu Awasthi, and Zeshan A. Chishti. Usimm : the utah simulated memory module. 2012.

[12] 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.

[13] E. Chen, D. Lottis, A. Driskill-Smith, D. Druist, V. Nikitin, S. Watts, X. Tang, and D. Apalkov. Non-volatile spin-transfer torque ram (stt-ram). In *68th Device Research Conference*, 2010.

[14] 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.

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

[16] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*, RecSys '16, page 191–198, New York, NY, USA, 2016. Association for Computing Machinery.

[17] Aditya Desai and Anshumali Shrivastava. The trade-offs of model size in large recommendation models : 100gb to 10mb criteo-tb dlrm model. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 33961–33972. Curran Associates, Inc., 2022.

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

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

[20] Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing*, STC '12, page 3–8, New York, NY, USA, 2012. Association for Computing Machinery.

[21] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas 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.

[22] Christopher W. Fletchery, Ling Ren, Xiangyao Yu, Marten Van Dijk, Omer Khan, and Srinivas 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.

[23] B. Gassend, G.E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and hash trees for efficient memory integrity verification. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pages 295–306, 2003.

[24] 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.

[25] 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.

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

[27] Paul Grubbs, Marie-Sarah Lacharite, Brice Minaud, and Kenneth G. Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 315–331, New York, NY, USA, 2018. Association for Computing Machinery.

[28] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1067–1083, 2019.

[29] Huifeng Guo, Wei Guo, Yong Gao, Ruiming Tang, Xiuqiang He, and Wenzhi Liu. Scalefreectr: Mixcache-based distributed training system for ctr models with huge embedding table. *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2021.

[30] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean Wu. Deeprecsys: a system for optimizing end-to-end at-scale neural recommendation inference. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 982–995. IEEE Press, 2020.

[31] Hanieh Hashemi, Wenjie Xiong, Liu Ke, Kiwan Maeng, Murali Annavaram, G. Edward Suh, and Hsien-Hsin S. Lee. Private data leakage via exploiting access patterns of sparse features in deep learning-based recommendation systems. In *Workshop on Trustworthy and Socially Responsible Machine Learning, NeurIPS 2022*, 2022.

[32] Wenpeng He, Fang Wang, and Dan Feng. H2oram: Low response latency optimized oram for hybrid memory systems. In *ICCD*, 2020.

[33] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 6th edition, 2019.

[34] Thang Hoang, Ceyhun D. Ozkaptan, Attila A. Yavuz, Jorge Guajardo, and Tam Nguyen. S^{3}oram: 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.

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

[36] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Network and Distributed System Security Symposium*, 2012.

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

[38] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1329–1340, New York, NY, USA, 2016. Association for Computing Machinery.

[39] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. Data recovery on encrypted databases with k-nearest neighbor query leakage. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1033–1050, 2019.

[40] Maximilian Lam, Jeff Johnson, Wenjie Xiong, Kiwan Maeng, Udit Gupta, Yang Li, Liangzhen Lai, Ilias Leontiadis, Minsoo Rhu, Hsien-Hsin S. Lee, Vijay Janapa Reddi, Gu-Yeon Wei, David Brooks, and G. Edward Suh. Gpu-based private information retrieval for on-device machine learning inference, 2023.

[41] Steven Lambregts, Huanhuan Chen, Jianting Ning, and Kaitai Liang. Val: Volume and access pattern leakage-abuse attack with leaked documents. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian D. Jensen, and Weizhi Meng, editors, *Computer Security – ESORICS 2022*, pages 653–676, Cham, 2022. Springer International Publishing.

[42] 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.

[43] Gang Liu, Kenli Li, Zheng Xiao, and Rujia Wang. Ps-oram: Efficient crash consistency support for oblivious ram on nvm. In *ISCA*, 2022.

[44] 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.

[45] 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.

[46] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013.

[47] Sujaya Maiyya, Seif Ibrahim, Caitlin Scarberry, Divyakant Agrawal, Amr El Abbadi, Huijia Lin, Stefano Tessaro, and Victor Zakhary. QuORAM: A Quorum-Replicated fault tolerant ORAM datastore. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3665–3682, Boston, MA, August 2022. USENIX Association.

[48] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie (Amy) Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 993–1011, New York, NY, USA, 2022. Association for Computing Machinery.

[49] 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.

[50] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems, 2019.

[51] G. Preethi, P. Venkata Krishna, Mohammad S. Obaidat, V. Saritha, and Sumanth Yenduri. Application of deep learning to sentiment analysis for recommender system on cloud. In *2017 International Conference on Computer, Information and Telecommunication Systems (CITS)*, pages 93–97, 2017.

[52] Rachit Rajat, Yongqin Wang, and Murali Annavaram. Pageoram: An efficient dram page aware oram strategy. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 91–107, 2022.

[53] Rachit Rajat, Yongqin Wang, and Murali Annavaram. Laoram: A look ahead oram architecture for training large embedding tables. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[54] Mehrnoosh Raoufi, Jun Yang, Xulong Tang, and Youtao Zhang. AB-ORAM: Constructing adjustable buckets for space reduction in ring oram. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 361–373, 2023.

[55] Mehrnoosh Raoufi, Jun Yang, Xulong Tang, and Youtao Zhang. EP-ORAM: Efficient nvm-friendly path eviction for ring oram in hybrid memory. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2023.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. Using address independent seed encryption and bonsai merkle trees to make secure processors os- and performance-friendly. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 183–196, 2007.

[61] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage. In *2016 IEEE Symposium on Security and Privacy (SP)*, 2016.

[62] Pavan Sanagapati. Ad display/click data on taobao.com. `https://www.kaggle.com/datasets/pavansanagapati/ad-displayclick-data-on-taobaocom`, 2024.

[63] 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.

[64] Elaine Shi, T-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious ram with o((logn)3) worst-case cost. *IACR Cryptol. ePrint Arch.*, 2011:407, 2011.

[65] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *2013 IEEE Symposium on Security and Privacy*, pages 253–267, 2013.

[66] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013.

[67] Jeffrey Stuecheli, Dimitris Kaseridis, David Daly, Hillery C. Hunter, and Lizy K. John. The virtual write queue: coordinating dram and last-level cache policies. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, page 72–82, New York, NY, USA, 2010. Association for Computing Machinery.

[68] 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.

[69] Jakub Szefer and Sebastian Biedermann. Towards fast hardware memory integrity checking with skewed merkle trees. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, 2014.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: A parallel oblivious file system. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012.

[76] H.-S. Philip Wong, Heng-Yuan Lee, Shimeng Yu, Yu-Sheng Chen, Yi Wu, Pang-Shiu Chen, Byoungil Lee, Frederick T. Chen, and Ming-Jinn Tsai. Metal–oxide rram. *Proceedings of the IEEE*, 2012.

[77] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. Overcoming the challenges of crossbar resistive memory architectures. In *HPCA*, 2015.

[78] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, 2015.

[79] 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.

[80] Chunxing Yin, Bilge Acun, Carole-Jean Wu, and Xing Liu. Tt-rec: Tensor train compression for deep learning recommendation models. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 448–462, 2021.

[81] 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.

[82] Xian Zhang, Guangyu Sun, Peichen Xie, Chao Zhang, Yannan Liu, Lingxiao Wei, Qiang Xu, and Chun Jason Xue. Shadow block: Accelerating oram accesses with data duplication. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 961–973, 2018.

[83] 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.

[84] Xiaoyu Zhang, Chao Chen, Yi Xie, Xiaofeng Chen, Jun Zhang, and Yang Xiang. A survey on privacy inference attacks and defenses in cloud-based deep neural network. *Computer Standards & Interfaces*, 83:103672, 2023.

[85] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, page 1059–1068, New York, NY, USA, 2018. Association for Computing Machinery.

[86] 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.

[87] Yu Zou and Mingjie Lin. Fast: A frequency-aware skewed merkle tree for fpga-secured embedded systems. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2019.