EFFICIENT SECURITY IN EMERGING MEMORIES

by

Joydeep Rakshit

B. Tech., National Institute of Technology, Durgapur, 2014

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2018

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This dissertation was presented

by

Joydeep Rakshit

It was defended on

June 11, 2018

and approved by

Kartik Mohanram, Ph.D., Associate Professor,

Department of Electrical and Computer Engineering

Jun Yang, Professor, Ph.D.,

Department of Electrical and Computer Engineering

Ervin Sejdić, Ph.D., Associate Professor,

Department of Electrical and Computer Engineering

Kevin Chen, Professor, Ph.D.,

Department of Electrical and Computer Engineering

Bruce Childers, Ph.D., Professor,

Department of Computer Science

Hai "Helen" Li, Ph.D., Associate Professor,

Department of Electrical and Computer Engineering, Duke University

Dissertation Director: Kartik Mohanram, Ph.D., Associate Professor,

Department of Electrical and Computer Engineering

Copyright © by Joydeep Rakshit
2018

EFFICIENT SECURITY IN EMERGING MEMORIES

Joydeep Rakshit, PhD

University of Pittsburgh, 2018

The wide adoption of cloud computing has established integrity and confidentiality of data in memory as a first order design concern in modern computing systems. Data integrity is ensured by Merkle Tree (MT) memory authentication. However, in the context of emerging non-volatile memories (NVMs), the MT memory authentication related increase in cell writes and memory accesses impose significant energy, lifetime, and performance overheads. This dissertation presents ASSURE, an Authentication Scheme for SecURE (ASSURE) energy efficient NVMs. ASSURE integrates (i) smart message authentication codes with (ii) multi-root MTs to decrease MT reads and writes, while also reducing the number of cell writes on each MT write.

Whereas data confidentiality is effectively ensured by encryption, the memory access patterns can be exploited as a side-channel to obtain confidential data. Oblivious RAM (ORAM) is a secure cryptographic construct that effectively thwarts access-pattern-based attacks. However, in Path ORAM (state-of-the-art efficient ORAM for main memories) and its variants, each last-level cache miss (read or write) is transformed to a sequence of memory reads and writes (collectively termed read phase and write phase, respectively), increasing the number of memory writes due to data re-encryption, increasing effective latency of the memory accesses, and degrading system performance. This dissertation efficiently addresses the challenges of both read and write phase operations during an ORAM access. First, it presents ReadPRO (Read Promotion), which is an efficient ORAM scheduler that leverages runtime identification of read accesses to effectively prioritize the service of critical-path-bound read access read phase operations, while preserving all data dependencies. Second, it presents LEO (Low overhead Encryption ORAM) that reduces cell writes by opportunistically decreasing the number of block encryptions, while preserving the se-

curity guarantees of the baseline Path ORAM. This dissertation therefore addresses the core challenges of read/write energy and latency, endurance, and system performance for integration of essential security primitives in emerging memory architectures.

Future research directions will focus on (i) exploring efficient solutions for ORAM read phase optimization and secure ORAM resizing, (ii) investigating the security challenges of emerging processing-in-memory architectures, and (iii) investigating the interplay of security primitives with reliability enhancing architectures.

TABLE OF CONTENTS

PREFACE xii
1.0 INTRODUCTION 1
1.1 Data integrity 1
1.1.1 Contributions 2
1.2 Data confidentiality 3
1.2.1 Contributions 5
1.3 Future Work 6
2.0 BACKGROUND AND MOTIVATION 8
2.1 Data integrity 8
2.1.1 Threat model 8
2.1.2 Encryption and authentication in NVMs 9
2.1.3 Motivation: Memory authentication overhead in NVMs 10
2.2 Data confidentiality 11
2.2.1 Threat model 12
2.2.2 Security definitions 12
2.2.3 Path ORAM 13
2.2.4 Motivation: ORAM integration overhead 15
3.0 ASSURE 17
3.1 Smart message authentication codes (SMAC) 17
3.1.1 SMAC: Observation 17
3.1.2 SMAC: Design 18
3.2 Multi-root Merkle Trees (MMTs) 19

3.2.1 Static multi-root Merkle Trees (SMMT) 20
3.2.1.1 SMMT: Observation 20
3.2.1.2 SMMT: Design 21
3.2.1.3 SMMT: Overhead 21
3.2.2 Dynamic multi-root Merkle Trees (DMMT) 22
3.2.2.1 DMMT: Observation 22
3.2.2.2 DMMT: Design 22
3.2.2.3 DMMT: Hot MBG prediction and update 23
3.2.3 ASSURE: Authentication architecture 25
3.2.3.1 Security 25
3.2.3.2 Logic overhead 25
3.2.3.3 Memory overhead 25
3.2.3.4 Latency overhead 26
3.3 Evaluation and results 26
3.3.1 Simulation framework 26
3.3.2 Workloads 27
3.3.3 Summary of results 27
3.3.4 NVM energy 28
3.3.5 Memory lifetime 29
3.3.6 System performance (IPC) 29
3.3.7 Sensitivity: n and P_{PRED} 30
3.4 Related Work 31
3.5 Conclusions 32
4.0 READPRO 33
4.1 Decomposing ORAM read latency 33
4.2 Inefficient write buffering in memory controller 33
4.3 ReadPRO architecture 36
4.4 Security of ReadPRO 38
4.5 Evaluation and Results 39
4.5.1 Methodology 39

4.5.2 ORAM read latency 40
4.5.3 Speedup 40
4.5.4 Effect of memory controller write buffer size 41
4.6 Related work 41
4.7 Conclusions 42
5.0 LEO 44
5.1 Write phase and encryption 44
5.2 LEO design 45
5.2.1 Observation 45
5.2.2 Design 45
5.2.3 Discussion: Partial line encryption vs LEO 47
5.2.4 Two-level counter design 50
5.2.5 Security: Overview 52
5.2.6 Security: Detailed discussion 52
5.2.7 Hardware overhead 54
5.3 Evaluation and Results 55
5.3.1 NVM energy and lifetime 56
5.3.2 System IPC 57
5.4 Related work 57
5.5 Conclusions 58
6.0 ORAM: DISCUSSION AND LOOKING FORWARD 59
6.1 Practicality and commercialization of ORAM 59
6.2 Tighter TCB with unsecure Operating System (OS) 61
6.3 Authentication/oblivious access in emerging hybrid memories 62
6.4 Authentication and obfuscation: A broader perspective 64
7.0 FUTURE WORK 66
7.1 Improving ORAM efficiency 66
7.2 Security for processing-in-memory architectures 68
7.3 Security-reliability co-design 68
BIBLIOGRAPHY 70

LIST OF TABLES

1 ASSURE evaluation: Workloads 27
2 Results: Summary 28
3 ReadPRO: Evaluation setup 39
4 LEO evaluation: Setup 55

LIST OF FIGURES

1 Bonsai Merkle Tree 10
2 Effect of authentication: Cell writes, NVM energy, and system IPC 11
3 Path ORAM: Illustration 14
4 SMAC: Illustration 19
5 SMMT: Design 21
6 DMMT: Design 23
7 DMMT: Hot MBG prediction architecture 24
8 ASSURE results: NVM energy 28
9 ASSURE results: Memory lifetime 29
10 ASSURE results: System performance 30
11 ASSURE results: Sensitivity analysis 31
12 ReadPRO: Illustration 34
13 ReadPRO: Architecture 37
14 ReadPRO results: Read Latency Reduction 40
15 ReadPRO results: Speedup 41
16 ReadPRO results: Write Buffer Sensitivity 42
17 LEO: Illustrative example 46
18 LEO: Eviction scenario 47
19 DEUCE security vulnerability 48
20 LEO security 49
21 LEO: Two-level counter design 50
22 LEO results: NVM energy reduction 56

23 LEO results: NVM lifetime improvement 57
24 LEO results: Speedup 58

PREFACE

I would like to take this opportunity for expressing my gratitude to everyone who has made this thesis possible. The single biggest influence on this work is my advisor, Prof. Kartik Mohanram. I am grateful for the academic and intellectual freedom that Kartik offered in pursuing my research, while providing essential mentoring on maintaining the highest standards of work in whichever domain I chose to pursue. Kartik's high standards of professionalism, attention to detail, and academic integrity have been the primary forces in shaping this dissertation to its present form, while also instilling these values in me for my future endeavors. Apart from his immense knowledge, Kartik has demonstrated by example how soft skills like representation of ideas matter in the field of research, and I hope to carry his teachings into my future.

I am thankful to my Ph.D. committee members—Prof. Jun Yang, Prof. Kevin Chen, Prof. Ervin Sejdić, Prof. Bruce Childers, and Prof Hai "Helen" Li for their motivation and helpful suggestions in shaping this thesis. I would like to thank Prof. Jun Yang for her valuable suggestions and discussions in her computer architecture course, which motivated me to take up this field for my research. Prof. Childers' course of advanced computer architecture provided a firm understanding of how to conduct and critique computer architecture research at the highest levels. Prof Li's course on hardware design methodologies was very helpful in providing a firm foundation of designing hardware modules from scratch, which I utilized later in my research to evaluate the hardware overheads of my security solutions.

This journey of four years would not have been as productive and enjoyable without the support of my colleagues from the Pitt Circuits and Systems Lab. First, I am particularly indebted to my colleague and housemate for 4 years, Shivam Swami, who started his Ph.D journey along with me. Apart from the intense discussions between us that arose from sharing the same research domain, his help and kind words and deeds have been very helpful in providing a comfortable atmosphere. I

am also thankful to my colleague Poovaiah M. Palangappa for strengthening my knowledge in my research field and in computing practices in general. Finally, I will fondly remember the various technical and fun discussions with my colleague Ali Al-Suwaiyan.

Last, but definitely not the least, I would like to acknowledge the support that I have received from my family during my Ph.D tenure. I would like to thank my parents, Sanjib and Krishna Rakshit, for their unwavering belief in my potential. They have made immense sacrifices for my education and to send me to this university, and this dissertation would not have been possible without their blessings. My grandparents, although no longer with me, have always encouraged me in my endeavors and my dreams, and I want to acknowledge their support and love to guide me through the formative years of my life and making me a better person. Finally, I would like to thank my girlfriend and partner, Ankita Bishayee, who has been with me in the highs and lows of my life as I was pursuing this doctorate. She has provided me strength when the going got tough, and amplified my successful moments by being a part of them. She made this difficult journey of 4 years smooth, enjoyable, and memorable.

1.0 INTRODUCTION

The wide adoption of cloud computing and storage is accompanied by a significant increase in data security challenges, exposing confidential data to multiple points of attack. Due to severe security implications, these emerging computing models have established data security as a first order design concern [1–4]. Similar to most computing security platforms, data security encompasses the three major pillars of integrity, confidentiality, and availability [5]. This dissertation focuses on architecting low overhead solutions to preserve integrity and confidentiality of data in emerging memory systems; data availability is ensured by orthogonal techniques [6–9]. The following sections briefly discuss the overheads of ensuring integrity and confidentiality of data in memory, and the contributions of this dissertation in addressing those overheads.

1.1 DATA INTEGRITY

Data integrity—a core component of the secure computation model—is the ability to detect adversarial tampering of (i) stored data in memory, or (ii) data transactions to/from memory. State-of-the-art memory authentication solutions maintain a logical data structure, Merkle Tree (MT), whose nodes are obtained by recursive computation of message authentication codes (MACs) over memory blocks, where MAC constitutes a cryptographic signature of the input data. In an MT, a parent node MAC ensures integrity of its child node MACs. MT memory authentication ensures the integrity of fetched (written) memory blocks by verifying (updating) its MAC lineage upto the MT root on the secure processor [1, 10–12].

MT memory authentication reduces system performance by introducing additional memory reads and writes, specifically to read/update the MT. Previous work have investigated and provided efficient architectural solutions to significantly reduce integrity-related memory traffic and thereby

the performance overhead [10–13]. However, most of the prior research was based on single or dual core systems with small DRAM-based main memory of $\leq$ 4GB. Current multicore systems enable execution of multiple concurrent applications, increasing the combined working set of the system and imposing an increasing demand on primary memory. Although DRAM-based memory has been the state-of-the-art primary memory for decades, the high energy consumption and poor scaling potential of DRAM has spurred research in emerging resistance-class non-volatile memories (NVMs) such as phase change memory (PCM) [14, 15], resistive RAM [16, 17], spin-transfer torque RAM (STT-RAM) [18], and 3D X-Point [19] because of their superior scalability, lower energy consumption due to non-existent refresh operations, and higher data density.

Although NVMs offer multiple benefits, the NVM write energy (latency) is higher in comparison to the read energy (latency), and also higher in comparison to DRAM write/read energy (latency) [20, 21]; these differences are exacerbated in multi-/triple-level cell (MLC/TLC) NVMs [22]. MT memory authentication ensures data integrity, it invariably incurs high NVM energy and performance penalty due to increased cell writes and memory accesses. The simulations of SPEC CPU2006 [23] workloads show that state-of-the-art MT memory authentication increases cell writes (NVM energy) to 5.8$\times$ (5.3$\times$) and degrades system IPC to 0.65$\times$ in comparison to a nominal encrypted triple-level cell (TLC) RRAM architecture. This motivates the re-evaluation of prior schemes in the context of NVM memory systems, and the development of low penalty solutions for NVM authentication.

1.1.1 Contributions

ASSURE: This dissertation presents an energy efficient **Authentication Scheme for SecURE** (ASSURE) NVMs, preserving the authentication properties of the underlying MT. ASSURE synergistically integrates smart message authentication codes (SMACs) and logically resizable multi-root MTs (MMTs) for low penalty memory authentication. SMAC leverages the observation that only the modified words are re-encrypted on consecutive write-backs to a memory location [24,25]. SMAC partitions the MAC at word-level granularity and recomputes only those MAC words corresponding to the re-encrypted words during a memory write.

ASSURE complements SMACs with multi-root MTs (MMTs) to reduce the memory accesses associated to MT authentication. MMTs maintain multiple smaller static/dynamic MTs, collec-

tively spanning the memory, with their roots on the secure processor. Static MMTs (SMMTs) partition the memory into memory block groups (MBGs) and statically maintain individual MTs for each MBG. Since each individual MT spans a smaller number of blocks, they have fewer levels, substantially reducing the number of MT levels read/updated for authentication. However, SMMTs incur significant processor-side on-chip storage for maintaining multiple MT roots. ASSURE proposes dynamic MMTs (DMMTs) for low storage overhead MMT, which leverage the spatial and temporal locality of memory accesses in practical workloads to dynamically predict the frequently (infrequently) accessed hot (cold) MBG(s). Hot (cold) MBG(s) are assigned to the smaller hot (larger cold) MT. Each parent SMAC in an MMT is partitioned into k words, where k is the order of the MMT, and only those words corresponding to modified children SMACs are recomputed; this extends the cell write reduction advantages of SMACs to MMT nodes by eliminating unnecessary parent MAC computation for unmodified children MACs.

ASSURE is evaluated on a TLC RRAM architecture for NVM energy, lifetime, and system IPC, and compared to state-of-the-art bonsai MT (BMT) [11] in the presence of two encryption frameworks: (i) dual counter encryption (DEUCE) [24] and (ii) Smartly EnCRypted energy EfficienT NVMs (SECRET) [25]. NVMain [26] is used to estimate NVM energy on memory access traces of SPEC CPU2006 benchmarks [23] generated using Intel Pin toolset [27]. The simulations show that on average, SMMT ASSURE (DMMT ASSURE) reduces NVM energy by 59% (55%) over BMT, because SMACs prevent redundant MAC recomputation of unmodified words and MMTs reduce MT accesses. The lifetime evaluations using an in-house lifetime simulator show that on average, SMMT ASSURE (DMMT ASSURE) improves memory lifetime by 2.36× (2.11×) over BMT due to significant cell write reduction. The full-system evaluations on MARSS [28] of composite SPEC CPU2006 workloads show that on average, SMMT ASSURE (DMMT ASSURE) improves system IPC by 11% (10%) over BMT.

1.2 DATA CONFIDENTIALITY

Data confidentiality is the capability to prevent unauthorized disclosure of plaintext data (i) stored in memory, or (ii) during data transactions to/from memory. Whereas data confidentiality is effectively ensured by encryption, the plaintext memory addresses and associated memory access

patterns can be exploited as a side-channel to obtain confidential data. Existing attacks reveal that memory access patterns can be leveraged to identify the control flow graph (i.e., behavior) of a program [29] or the associated encrypted data/encryption key [30–32]. Oblivious RAM (ORAM) is a cryptographic primitive that effectively obfuscates the memory access pattern, preventing information leakage about (i) the address accessed, (ii) the access type (read or write), and (iii) the data being read/written [33–35]. ORAM primarily functions by encrypting the data and continuously shuffling their memory locations. Recent research widely adopts the Path ORAM construct for its efficiency and algorithmic simplicity [3, 4, 31, 34–40].

Path ORAM: In Path ORAM, the memory is organized as a binary tree, wherein each node, termed a bucket, contains a fixed number of slots to store encrypted data blocks. The logical address (program/virtual address after page table translation) of the data blocks is randomly mapped to one of the leaves (i.e., assigned a LeafID) and the data block can be stored in any bucket on the path from the root to the mapped leaf. The LeafIDs of data blocks are contained in a data structure called the *PosMap*. A logical address (LA) access (read **or** write) to the Path ORAM after a last-level cache (LLC) miss is composed of two phases. In the *read phase*, encrypted blocks from every bucket on the mapped path are fetched, decrypted, and the target data block remapped to a new LeafID and sent to (updated by) the processor if it was an LA read (write) access. In the *write phase*, all possible fetched blocks are re-encrypted and evicted back to the path, placing them in buckets as close to the leaves as possible. Path ORAM derives its obfuscation guarantees primarily from the random leaf remapping of the accessed address during the read phase; therefore, a new path is accessed each time a particular LA is requested from ORAM.

The primary overhead of ORAM integration with main memories is the expansion of a single memory access (LA read or LA write) to multiple memory reads **and** writes, which amplifies memory traffic, thereby increasing effective memory access latency, increasing memory energy footprint, and degrading overall system performance; additionally, ORAM significantly deteriorates memory lifetime for NVM-based memory systems due to the increased write frequency for each logical memory access. This dissertation follows a holistic approach of reducing the ORAM overheads by providing architectural solutions for improving both read phase and write phase performance of state-of-the-art ORAM.

1.2.1 Contributions

ReadPRO: First, this dissertation focuses on optimizing the read phase of an ORAM access, specifically reducing the latency of LA read accesses, which stalls the application thread execution until the requested data is received from memory. The read phase of an LA read access is on the critical path of program execution, since the processor is waiting for the target data to be fetched from the memory.The key *observation* is that due to in-order scheduling of the LA accesses in baseline ORAM controller, the critical-path-bound read phase operations of LA read accesses are delayed by pending read and write phase operations of older LA write accesses, which are not on the critical path of program execution. Previous work like Fork Path [38] and Tiny ORAM [41] reduce the number of block writes to decrease overall ORAM access latency; however, this contribution, ReadPRO (**Read** **Pro**motion) is the first work to explicitly address the critical-path-bound feature of read access read phase operations to reduce ORAM read latency and improve system performance.

ReadPRO is an ORAM scheduler for dynamic threshold-bound prioritization of LA read accesses at the last-level cache (LLC) and ORAM controller interface, ensuring faster performance-critical data fetches while preserving all data dependencies and incurring low logic and memory overhead. ReadPRO is applicable to both ORAM in DRAM-based and NVM-based main memories. For a conservative evaluation of the advantages of ReadPRO over state-of-the-art ORAM architectures, full-system evaluations are performed by integrating a DRAM-based memory system across SPEC CPU2006 benchmarks; ReadPRO decreases the average ORAM read latency by $4\times$, improving system performance by 38%.

LEO: Second, this dissertation focuses on architectural solutions to optimize the write phase of an ORAM access, specifically targeting the NVM systems due to their high write cost (latency, energy, performance). The re-encryption of all the blocks on a path during the write phase of an ORAM access increases the cell-write rate, since the diffusion property of encryption algorithms results in a 50% bit-flip rate on average during a memory write [24, 25]. Although this increased cell-write rate is not a dominant concern for DRAM, it adversely impacts NVM energy and lifetime because of the high overhead of NVM writes over NVM reads and DRAM read/writes [42, 43], motivating the development of low energy, high lifetime NVM Path ORAM. To address this challenge, this

dissertation proposes LEO (**<u>L</u>ow overhead <u>E</u>ncryption <u>O</u>RAM**), which is a secure and optimal encryption framework for NVM Path ORAM. LEO minimizes the redundant re-encryption of unmodified blocks during the write phase of a Path ORAM access, decreasing NVM cell writes, and its corresponding overheads during the write phase of an ORAM access.

LEO reduces redundant re-encryptions securely by mandating that all buckets along an accessed path should experience block re-encryptions equal to the highest count of new/modified blocks written to an individual bucket on that path during an ORAM access. In the buckets, the new/modified blocks, and if required, some randomly selected unmodified blocks are re-encrypted to achieve uniform re-encryption count across all buckets; the remaining unmodified blocks are not re-encrypted. For this optimized encryption, LEO utilizes a two-level counter architecture for counter-mode encryption (CME) of the blocks. LEO preserves the security of the encryption architecture in the baseline Path ORAM [34] and does not leak any additional information beneficial to the adversary. Since LEO is geared towards reducing bit flips on each ORAM access in NVMs, LEO is evaluated on a single-level cell (SLC) PCM architecture using workloads from SPEC CPU2006 [23] benchmark suites. The simulations on single-level cell (SLC) PCM architecture using SPEC CPU2006 [23] benchmarks show that LEO decreases average NVM energy by 60%, enhances lifetime by 1.51×, and improves system performance by 9% over the state-of-the-art ORAM architectures.

1.3 FUTURE WORK

Future research directions will focus on (i) exploring efficient solutions for ORAM read phase optimization and secure ORAM resizing, (ii) investigating the security challenges of emerging processing-in-memory architectures that require plaintext data on the memory modules for processing purposes, and (iii) investigating the interplay of security primitives with reliability enhancing architectures, focusing on leveraging the reliability improvement techniques to efficiently reduce security-related overheads.

First, although ReadPRO decreases the ORAM read latency, it is still considerably ($\geq 2\times$) higher than a system that does not employ ORAM for access pattern obfuscation. In an ORAM primitive with ReadPRO scheduling, each logical memory access is still amplified to multiple memory read **and** write operations (i.e., the read and the write phase, respectively). Therefore,

there is a large scope of improvement in ORAM performance by further optimization of the critical-path-bound read phase of an ORAM access. Additionally, the memory traffic, i.e., the number of blocks read/written back in an ORAM access depends on the number of levels in the tree; therefore, it can be reduced by dynamically decreasing the size of the ORAM tree. Although the dynamic tree sizing solutions developed in ASSURE solves a similar problem for MT memory authentication, it assumes memory access locality in the external memory; an ORAM is specifically designed to obfuscate memory access pattern, and thereby eliminate any memory access locality to the external unsecure memory. This motivates the development of a secure ORAM tree resizing algorithm and architecture to dynamically scale memory traffic.

Second, modern applications in domains like machine learning, graph processing, and other similar fields that are inherently parallel, utilize large datasets, and require high bandwidth memories are strong candidates for processing-in-memory (PIM) architectures [44–57]. PIM architectures generally constitute logic capabilities on the memory itself, thereby utilizing the high on-DIMM bandwidth of the memory. However, since PIM performs processing on the memory DIMM, it requires the data to be in plaintext in the external memory. Hence, enabling PIM will lead to prevention of storing encrypted data on memory, leading to data confidentiality challenges; alternatively, homomorphic encryption [58] can be implemented, which enables processing on encrypted data, albeit with orders of magnitude higher latency. This motivates a detailed investigation of security in PIM architectures, and development of efficient security solutions to prevent a reduction in the benefits of PIM.

Third, recent work has shown that the co-design of security and reliability constructs improves the system performance by utilizing authentication constructs to perform efficient error detection in DRAM-based systems [59]. However, [59] is developed around ECC-based reliability systems in DRAM memories. The error characteristics in NVM-based systems is significantly different than DRAM-based systems, which motivated the development of a vast array of NVM reliability improvement techniques over the past few years [21, 60–69]. This necessitates a detailed study on the interplay of reliability improvement solutions for NVM systems and associated security, thereby developing insights for architecting efficient reliable and secure NVM systems.

2.0 BACKGROUND AND MOTIVATION

This chapter discusses the threat model considered in this work, the state-of-the art solutions for data integrity and data confidentiality, and the impact of these solutions on the memory system performance, energy, and lifetime parameters. Note that in this dissertation, data integrity and data confidentiality have progressively stringent threat models, and therefore different primary security primitives and challenges for integration with memory systems. The relevant details for data integrity and data confidentiality are presented separately in the following sections.

2.1 DATA INTEGRITY

This section provides a detailed discussion on the threat model, state-of-the-art memory authentication primitive, and the overheads of ensuring data integrity through memory authentication when integrated with emerging NVM memories.

2.1.1 Threat model

An ideal secure computing platform requires three cornerstone properties: (i) confidentiality, (ii) integrity, and (iii) availability [5]. However, system design simplification and feasibility requires a specific threat model that differentiates the threats that the system protects against, and those not considered as part of the model [5]. For data integrity, this dissertation considers a threat model encompassing attacks on data confidentiality and data integrity; the trusted computing base (TCB) consists of the processor and core parts of the operating system (e.g., security kernels), whereas the off-chip memory and the processor-memory bus are untrusted [10–12]. Data confidentiality attacks aim to obtain secret data stored in memory or data being transfered to/from memory, motivating memory encryption. However, encryption does not protect against integrity attacks, where the

adversary alters the data stored in or being transfered to/from memory. Data integrity attacks can be categorized into spoofing, splicing, and replay attacks [10, 11]. In spoofing attacks, the adversary replaces an existing valid memory block with fake data. In splicing attacks, the attacker swaps the memory content between two locations. Finally, in replay attacks, the content of a memory location is reverted back to an older value.

It is widely accepted that data integrity attacks can be thwarted by memory authentication, which verifies the integrity of all off-chip communications to/from the secure processor [10–12]. To prevent both integrity and confidentiality attacks, memory authentication must be deployed concurrent with memory encryption.

2.1.2 Encryption and authentication in NVMs

Memory encryption is achieved by applying a block cipher over plaintext data [10, 24, 25, 70]. Prior works advocate the use of counter-mode encryption (CME) to offset the latency of encryption/decryption during memory write/read [10, 24, 25, 70]. In CME, the data is XORed with a one-time pad (OTP) generated using a block cipher, which includes a secret key and a seed as inputs for encryption. The seed is composed of the memory block address and an associated counter that increments on each memory write. However, the diffusion property [71] of encryption drastically increases cell writes, which is especially undesirable in NVMs due to their high write energy/latency [24]. DEUCE [24] reduces the cell writes by re-encrypting only the modified words on memory writes; SECRET [25] further improves encryption in MLC/TLC NVMs by preventing zero-word re-encryption and XOR-based energy masking.

Memory authentication of an encrypted NVM system guarantees the integrity of both encrypted data and the counters of CME, since tampering of either results in the generation of invalid plaintext during decryption. Prior works have advocated the use of Merkle Tree (MT) memory authentication, which is proven to be secure against spoofing, splicing, and replay attacks [10–12]. State-of-the-art memory authentication schemes use keyed hash message authentication codes (HMACs), which utilize a cryptographic hash function (e.g., SHA-1) and a secret key to generate a hash signature that includes the data block along with its corresponding counter and line address as inputs [11, 12]. MT memory authentication maintains a hierarchical tree struc-

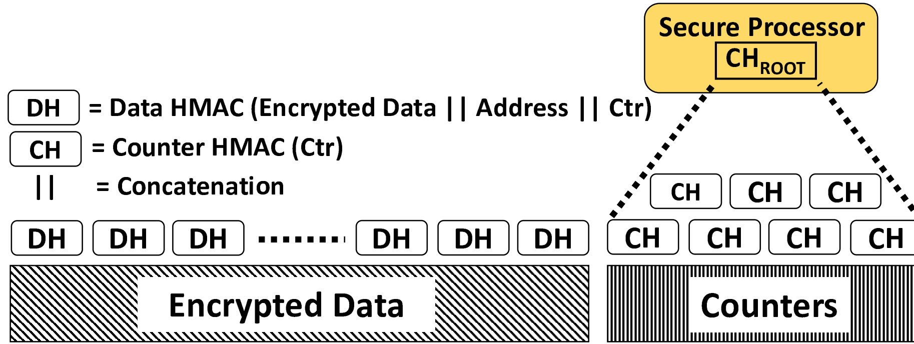

Figure 1: **State-of-the-art bonsai MT (BMT): BMT maintains an MT over the counters, and a single layer of HMACs over encrypted data. The data HMACs include the encrypted data with its line address and counter as input, protecting encrypted data from spoofing (data), splicing (address), and replay (counter) attacks.**

ture of these HMACs, with the data and counter as its leaf nodes. In an MT, each parent node is an HMAC signature of its children node HMACs (data/counter in case of leaf nodes). The secret HMAC key and the MT root is stored on the secure, tamper-proof processor, preventing spoofing, splicing, or replay attacks. During reads, the memory block integrity is ascertained by verifying its HMAC lineage upto the MT root; in contrast, writes result in recomputation of the HMAC lineage upto the MT root to reflect the new data.

The state-of-the-art MT memory authentication architecture is the bonsai MT (BMT) [11], illustrated in Fig. 1. BMT leverages the CME architecture and maintains an MT over only the counters of a memory line rather than over both counters and data. BMT keeps a single level of HMAC over the data memory, where the data HMACs use the encrypted data block, address, and the counter as input. Although the encrypted data is not protected by an MT, it is immune to replay attacks because the data HMACs include BMT-protected counters as input. Since the counter memory is considerably smaller than the data memory, the BMT has significantly fewer levels than an MT over both data and counters, reducing the MT reads/writes and improving system IPC. Without loss of generality, BMT is referred to as MT for the rest of the dissertation.

2.1.3 Motivation: Memory authentication overhead in NVMs

The NVM write energy (latency) is higher in comparison to the read energy (latency), and also higher in comparison to DRAM write/read energy (latency) [20, 21]; these differences are exac-

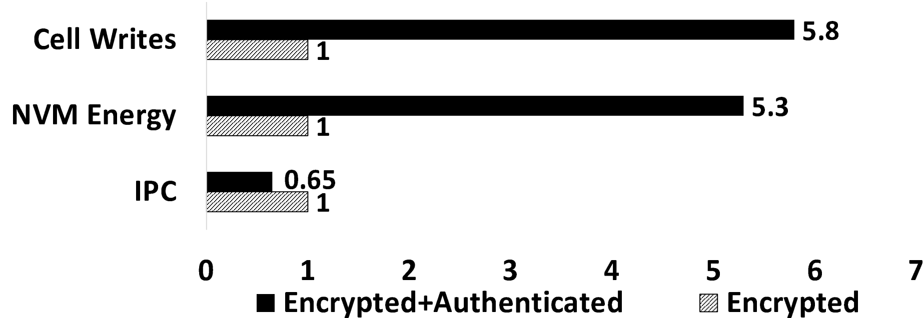

Figure 2: **Comparison of average cell writes, NVM energy, and system IPC (norm. to baseline) of encrypted RRAM baseline with an encrypted RRAM deploying memory authentication, Memory authentication increases the cell writes (NVM energy) to 5.8× (5.3×), and degrades system IPC to 0.65× in comparison to the baseline.**

erbated in multi-/triple-level cell (MLC/TLC) NVMs [22]. HMACs demonstrate strong diffusion property, similar to data encryption [71], resulting in a high cell write rate, and render NVM write-reduction techniques like [72, 73] ineffective in practice. Hence, the data and counter HMACs in the BMT incur significant NVM energy/latency overhead and lifetime reduction. Further, additional memory accesses are introduced when the counter MT is read/updated on each counter read/write. These reads/writes integral to memory authentication stall critical data/counter reads or writes to the same memory bank, degrading system IPC. In Fig. 2, simulations of SPEC CPU2006 workloads to illustrate that BMT authentication over a DEUCE-encrypted TLC RRAM increases the cell writes (NVM energy) to 5.8× (5.3×) and degrades system IPC to 0.65× in comparison to an unauthenticated DEUCE-encrypted TLC RRAM.

In summary, whereas memory authentication is indispensable to a secure computing platform, it degrades NVM energy, latency, lifetime, and system IPC. ASSURE reduces authentication cell writes and memory accesses for a low penalty NVM authentication solution.

2.2 DATA CONFIDENTIALITY

This section provides a detailed discussion on the threat model for access-pattern-based attacks on data confidentiality, state-of-the-art access-pattern-obfuscation solution (i.e., ORAM), and the overheads of ensuring data confidentiality through ORAM integration.

2.2.1 Threat model

Following previous work on ORAM [31, 35–39, 74], this dissertation adopts the secure processor paradigm in threat modeling, wherein the trusted computing base (TCB) is comprised of the processor and all on-chip data, while the off-chip memory and the processor-memory bus are untrusted. The adversary is capable of monitoring information (data, address, and command) on the memory bus and from the external memory (i.e., the DIMM). Although the data is encrypted, the attacker can analyze the plaintext addresses and commands, i.e., the attacker can analyze the memory access patterns to expose confidential information about the encrypted data [29–32]. ORAM effectively conceals the access-pattern-based side-channel by cryptographically obfuscating the memory access patterns [31,35–39]. Note that information leakage through ORAM timing-channels such as ORAM access frequency or the program termination time is not consider. This information leakage can be eliminated/reduced by solutions proposed in [75], which are orthogonal to and integrable with the solutions proposed in this work (i.e., ReadPRO and LEO).

2.2.2 Security definitions

This section discusses the ORAM security definition from the secure processor perspective. The primary objective of ORAM is to completely hide or obfuscate the memory access pattern of the active applications from the adversary beyond the secure processor boundary. From the perspective of the attacker, two memory access patterns of the same length, but originating from different processes/threads on the processor core must be indistinguishable. To elaborate on the security guarantees of an ORAM construct, it should prevent the following information leakage about the memory access pattern [34]: (i) the plaintext data for being read/written, (ii) the memory operation (i.e., whether the memory access is a read/write), (iii) the actual address being accessed, (iv) the frequency at which a memory address is accessed, and (v) whether similar memory addresses are being accessed within a memory access sequence (i.e., linkability between two individual memory accesses).

For a more formal definition: Let

$$\vec{s} := \{(addr_N, op_N, data_N), ..., (addr_1, op_1, data_1)\}$$

represent a memory request sequence from the processor chip, where $addr_i$, op_i, and $data_i$ refer to the address accessed, the type (i.e., read/write), and the data read/written in the ith memory

access. Let $\hat{M}$ ($\vec{s}$) denote the memory access pattern observed by the external unsecure memory for the sequence of memory requests $\vec{s}$. For a memory system implementing ORAM, $\hat{M}$ ($\vec{s_a}$) and $\hat{M}$ ($\vec{s_b}$) for $a \neq b$ should be computationally indistinguishable and statistically independent, given $\vec{s_a}$ and $\vec{s_b}$ are of the same length.

2.2.3 Path ORAM

Secure processor design extensively utilize Path ORAM or its variants for memory access pattern obfuscation, primarily due to its efficiency and algorithmic simplicity [3, 4, 31, 34–39, 41, 75]. Path ORAM (ORAM henceforth) has two components: (i) the untrusted external main memory and (ii) the trusted on-chip ORAM controller.

The untrusted main memory is logically organized as a binary tree, as shown in Fig. 3. The ORAM tree levels range from 0 (root) to L (leaves), resulting in a tree with L+1 levels. The path from the root to a leaf l is denoted as path-l. Each node in the ORAM, termed a bucket, has a fixed number of slots, Z, to store data blocks, i.e., cache lines [36]; a slot can hold a data or a dummy block. Additionally, each node maintains a meta-data block consisting of the LAs of the Z blocks, mapped leaf IDs, and an encryption counter. Every block, except the counter, is kept encrypted in memory. The maximum percentage of ORAM blocks that can hold real data is termed ORAM utilization [36–38, 74, 76, 77].

The trusted ORAM controller has two parts: a *Frontend* and a *Backend*. The Frontend comprises of the LA queue, position map (PosMap), and address logic, whereas the Backend comprises of the stash, and encryption/decryption units. The LA queue at the interface of the LLC and the ORAM controller buffers the LA accesses from the LLC for dispatch to the ORAM [4, 38, 75]. The PosMap associates the LAs of data blocks to their corresponding leaf labels, i.e., LeafIDs. The address logic generates the addresses of the nodes on a path that are subsequently received by the memory controller on the processor to perform the read and write phase operations [37, 38]. The stash is a small on-chip storage for data blocks read from the ORAM tree, post decryption. The encryption/decryption units encrypt/decrypt data and meta-data blocks using a standard algorithm (e.g., AES) [78].

ORAM guarantees the following invariant by construction: A data block, mapped to a leaf l, is either in a bucket on the path-l **or** in the stash [34]. For cache misses, the LLC queries the ORAM

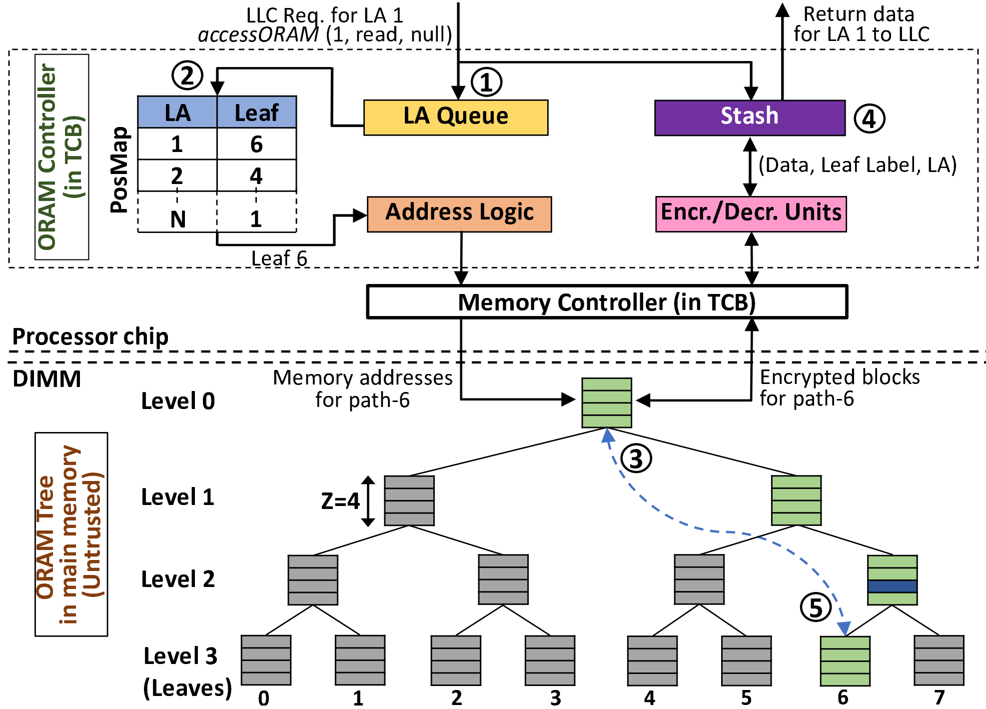

Figure 3: **An ORAM illustration with L=3 and Z=4. The LLC accesses data at LA 1, mapped to leaf 6. All the blocks on path-6 are fetched, decrypted, and the data blocks stored in the stash. Block 1, which was at the level 2 bucket, is sent to the LLC. Finally, path-6 is written back with as many blocks evicted from the stash as possible.**

controller through an accessORAM (*addr, op, data*) interface, where *addr* is the LA of the target data block, *op* refers to read/write (i.e., whether the LLC miss was a read or write), and *data* is the new data if *op* is a write (*data* is null when *op* is a read). The query is then transformed to an ORAM access by the ORAM controller as follows (illustrated in Fig. 3 for LLC read miss on LA 1): ① The stash is searched for a block with LA 1. If found, it is forwarded to the LLC; else the LA is pushed into the LA queue. ② The entry for LA 1 is queried in the PosMap, where it is mapped to leaf 6. Then, LA 1 is remapped to a random leaf l_{NEW} (not shown in the figure). ③ The memory controller initiates the *read phase*, wherein it fetches all blocks from path-6 to the processor and forwards them to the encryption/decryption units. The data blocks are decrypted and stored in the stash. ④ If *op* is a read, the data from the block with LA 1 is returned to the LLC; if *op* is a write, the accessed block is overwritten in the stash with the new data. ⑤ The *write phase* is completed, wherein all possible blocks are evicted from stash to path-6, pushing the blocks as close to the leaves as possible; additionally, all evicted and dummy blocks on the path are re-encrypted [36].

The data to be forwarded to the LLC for an LA read access is fetched from the memory during the read phase of the corresponding ORAM access. *ORAM read latency* is the time required for the ORAM controller to return the target data block after an LLC read miss.

Path ORAM: Security

The security of ORAM and its obfuscation property stems from the random leaf re-mapping of the logical addresses (i.e., the actual memory addresses). The PosMap lookup of an address $addr_i$ provides the leaf label l_i, where l_i=PosMap[$addr_i$] . Now, for an access pattern $\vec{s}$, where

$$\vec{s} := \{(addr_N, op_N, data_N), ..., (addr_1, op_1, data_1)\}$$

the adversary observes a path access sequence $\hat{P}$, where

$$\hat{P} := \{l_N, ..., l_1\}$$

Note that in this sequence, l_i is statistically independent l_j, where $i \neq j$; this is because each address is randomly mapped to a leaf label, and remapped when accessed. Therefore, even if $addr_i$=$addr_j$ (where $i \neq j$), $l_i \neq l_j$, i.e., even if the same address is accessed more than once in an memory request sequence, statistically independent paths are accessed on each iteration. Hence, the sequence of path accesses, $\hat{P}$, is indistinguishable from a randomly generated string of numbers. The attacker cannot infer any useful information about the true memory access pattern from $\hat{P}$, thereby obfuscating the access pattern efficiently.

Additionally, all the blocks in an ORAM is encrypted with a secret key stored on the processor, so the adversary cannot decipher the plaintext. The data integrity is ensured in ORAM by MT authentication, utilizing the inherent binary tree structure of an ORAM [37, 76].

2.2.4 Motivation: ORAM integration overhead

The translation of a single LA access to an ORAM access, which is a sequence of memory reads (read phase) and writes (write phase), increases the effective cache miss penalty, thereby degrading system performance (i.e., system IPC). The simulation results with a DDR3 DRAM system demonstrate that on average, ORAM integration increases LA read latency by 10×, decreasing system IPC by 4×. These considerations motivate a solution to improve the latency of critical-path-bound read access read phase operations in ORAM, enhancing system IPC.

Additionally, the mandatory re-encryption of all the blocks on a path during the write phase of an ORAM access increases the number of write operations. As discussed in Section 2.1.3, increased writes is not a dominant concern in DRAM; however, it amplifies NVM energy and reduces lifetime and performance because (i) NVM write energy and latency is higher than DRAM write/read energy and latency and (ii) NVM lifetime is orders of magnitude lower in comparison to DRAM [43]; these differences are exacerbated in multi-/triple-level cell (MLC/TLC) NVMs [43]. The simulation results on an SLC PCM system demonstrate that on average, ORAM integration increases NVM energy by 45$\times$, while reducing lifetime and system IPC by 40$\times$ and 10$\times$, respectively. These considerations motivate the development of a secure low-overhead encryption architecture for NVM ORAM efficiency.

Note that the architectural ORAM solutions (ReadPRO and LEO) developed in this work are agnostic to the underlying memory technology. However, these schemes, specifically the write reduction solution (LEO) are more suitable for NVM memory systems. LEO can also be integrated with DRAM-based memory system to reduce the memory write traffic, freeing up memory bandwidth for read phase of the subsequent path access, thereby improving read latency. However, the NVM memory-based evaluation of ReadPRO and DRAM memory-based evaluation of LEO will be done in future work.

To summarize, ORAM integration is essential to thwart access-pattern-based data confidentiality attacks; however, ORAM incurs significant overheads in data read latency, memory energy, memory lifetime (for NVMs), and system performance. The architectural solutions proposed in this dissertation, ReadPRO and LEO, reduce the data read latency and memory write traffic to improve system performance and memory lifetime, while reducing memory energy.

3.0 ASSURE: AUTHENTICATION SCHEME FOR SECURE ENERGY EFFICIENT NON-VOLATILE MEMORIES

This chapter describes ASSURE, a low overhead NVM authentication solution that deploys (i) smart MACs to reduce cell updates on authentication-related memory writes and (ii) multi-root MTs to reduce memory accesses for MT authentication. ASSURE preserves the security properties of classical MT authentication and is compatible with all NVM encryption solutions. Without loss of generality, DEUCE [24] is considered over SECRET [25] for NVM encryption in the discussions, because of its simpler architecture.

3.1 SMART MESSAGE AUTHENTICATION CODES (SMAC)

Without exception, hashed message authentication codes (HMACs) are the primary units of memory authentication [10–12]. However, HMACs incur increased (decreased) write energy (lifetime) owing to a high cell write rate (refer Sec. 2.1.3). This chapter proposes smart message authentication codes (SMACs) as a solution to realize security equivalent to HMACs with reduced (improved) write energy (lifetime) through decreased cell writes.

3.1.1 SMAC: Observation

Memory authentication must be integrated with memory encryption for secure, tamper-resistant memory content, as discussed in Sec. 2.1.2. State-of-the-art NVM encryption [24, 25] performs selective re-encryption of only the modified words to generate new ciphertext on each memory write. However, classical HMAC computation does not exploit this partial re-encryption, integral to efficient NVM encryption, and requires HMAC recomputation of the entire encrypted cache line; this results in redundant HMAC recomputation of the unmodified words.

3.1.2 SMAC: Design

The core advantage of SMACs over classical HMACs is that SMACs perform selective HMAC recomputation of the encrypted data by leveraging the partial re-encryption property of the underlying NVM encryption architecture. DEUCE partitions the cache line into words of equal width, and re-encrypts only the modified words during a memory write. SMAC partitions the HMAC at word-level granularity and recomputes only those words corresponding to the re-encrypted words during a memory write; this eliminates cell writes due to the redundant HMAC computation of unmodified words.

To achieve selective HMAC computation of only the modified words, SMAC splits the original encrypted cache line into two decoupled intermediate messages (IMs) corresponding to the modified and unmodified words. The IMs have the same length and partition boundaries as the encrypted cache line. The first (second) IM, IM_1 (IM_2) is constructed from the modified (unmodified) words with the unmodified (modified) words zeroed out. IM_1 (IM_2) is then provided as input to a keyed cryptographic hash function, generating the intermediate HMACs, IH_1 (IH_2). Similar to IMs, the IHs are also partitioned at word-level granularity. The final HMAC (FH) is constructed with IH_1 (IH_2) words for the corresponding modified (unmodified) word positions. For example, if word k of an encrypted cache line is modified (unmodified), word k of FH is constituted by word k of IH_1 (IH_2).

During a read, the SMAC requires meta-data to identify the modified/unmodified word positions of the previous write to an address for valid FH reconstruction of the fetched encrypted data. Note that the underlying DEUCE architecture records modified bits (modbits) to track the modified/unmodified words. SMAC leverages the DEUCE modbits for tracking modified/unmodified words, incurring zero memory overhead (Note that ASSURE provisions independent modbits if implemented over SECRET [25], since SECRET does not provision modbits). Since valid decryption in DEUCE and FH reconstruction in SMAC depends on the critical modbits, modbit integrity protection is proposed through modbit inclusion in the original input assignment to IM_1. The modbits assigned to IM_2 are always zeroed out, since IM_2 represents the unmodified words of the cache line. Due to the strong diffusion property of HMAC algorithms, any change in modbits is reflected in subsequent alteration of the FH words corresponding to the modified word positions, enabling modbit integrity protection.

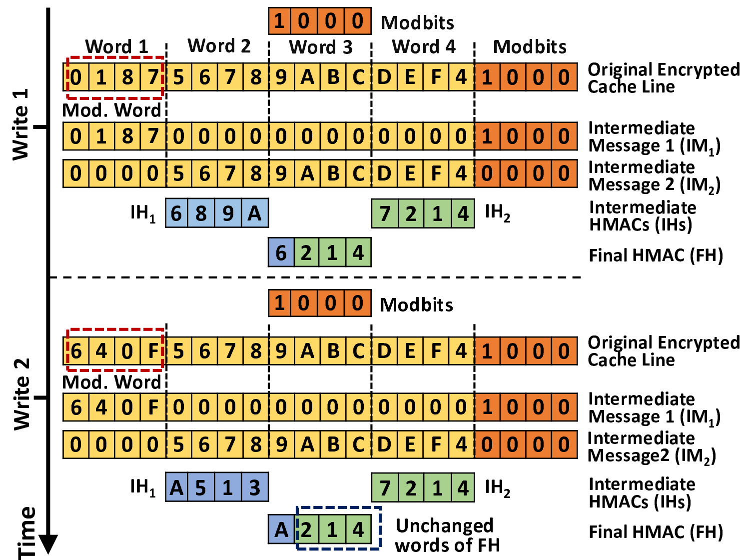

Figure 4: **SMACs eliminate cell writes for unmodified words (2, 3, and 4) during HMAC computation. Between writes 1 (W_1) and 2 (W_2), word 1 gets modified in the original encrypted cache line, altering the intermediate HMAC 1 (IH_1), resulting in modification of word 1 in the final HMAC (FH); however, IH_2 is unaltered due to unmodified words 2, 3, and 4, eliminating redundant cell writes for unmodified words 2, 3, and 4 of the FH.**

Figure 4 illustrates SMACs over a sequence of 2 consecutive writes. Without loss of generality, consider a 64-bit encrypted cache line with a word size of 16 bits (represented in hexadecimal), and one modbit per word. For write 1 (W_1), only word 1 is modified, setting the modbit for word 1. IM_1 is given by the modified word 1 and the modbits, with the unmodified words 2, 3, and 4 zeroed. IM_2 is given by the unmodified words 2, 3, and 4, with the modified word 1 and the modbits zeroed. The IHs are generated by treating the IMs as inputs for the cryptographic hash function, with the FH obtained by selecting word 1 from IH_1 and the rest from IH_2. On write 2 (W_2), word 1 is modified again, subsequently altering IM_1 and IH_1, resulting in a modification to word 1 of FH. However, words 2, 3, and 4 of the FH are unmodified at W_2 due to the unmodified words 2, 3, and 4 in the original encrypted cache line, decreasing (increasing) NVM write energy (lifetime).

3.2 MULTI-ROOT MERKLE TREES (MMTS)

In MT authentication, a single MT spans the counter memory with a single root on the secure processor. However, MT authentication incurs a high penalty of additional memory reads (writes)

to fetch and verify (update) the corresponding MT branch of a read (written) counter memory block, degrading NVM energy and system IPC. This chapter proposes multi-root MTs (MMTs) that maintain multiple smaller MTs having fewer levels (with multiple corresponding roots on the secure processor) as a novel alternative to the classical single-root MT. The multiple roots of the MMT collectively span the entire counter memory, with each MT assigned to one memory block.

Although static MMTs achieve substantial MT read/write reduction, they incur high secure processor-side storage overhead of multiple roots. ASSURE leverages the spatial and temporal locality of memory accesses in practical workloads to realize a prediction architecture that dynamically identifies and maintains a smaller MT over the frequently accessed memory block, while spanning all other memory blocks with a larger MT. This reduces storage overhead to only 2 roots on the processor.

3.2.1 Static multi-root Merkle Trees (SMMT)

This section begins with a discussion of the static MMT (SMMT) architecture. In this approach, MTs are statically assigned to groups of memory blocks and maintain their roots on the secure processor, reducing MT traversal levels, thereby improving NVM energy and system IPC.

3.2.1.1 SMMT: Observation In MTs, the leaf nodes represent the integrity-preserved memory blocks. SMMTs leverage the observation that a smaller MT that spans fewer leaf nodes (memory blocks) is composed of fewer MT levels, reducing the reads (writes) to verify (update) a corresponding MT branch during a leaf node read (write). This observation is illustrated using Fig. 5. Figure 5(a) represents a classical single-root MT spanning 8 leaf nodes ($L_0 - L_7$), with the authentication path for leaf L_1 highlighted. The following steps are executed for authenticating L_1 during a read: First, L_0 and L_1 are read and hashed together with the resultant hash (HMAC) compared to MT node M_{10}. Second, M_{10} and M_{11} are hashed together, with the resultant hash compared to M_{20}. The recursive read, hash, and compare operations continue until the root R. In Fig. 5(b), the leaf nodes are alloted to 2 equal groups (G_0 and G_1), with an independent MT spanning each group and their roots (R_0 and R_1) maintained on the secure processor. As evident from the highlighted path, a smaller MT results in 1 less MT level read/write for authentication of L_1. Generally, a k-ary MT with n leaf node groups achieves $\log_k n$ MT level reduction.

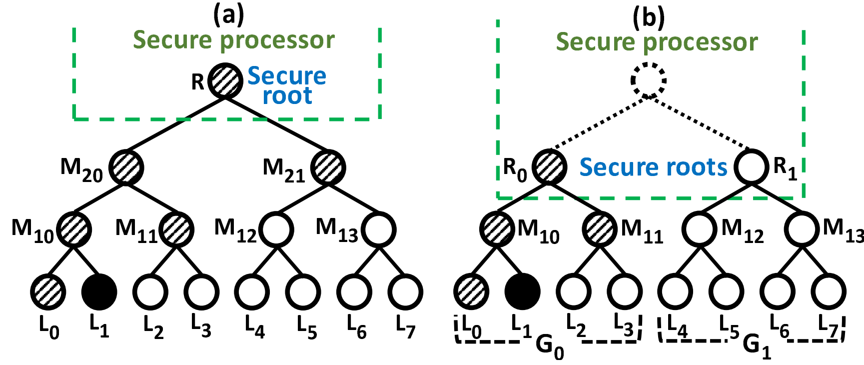

Figure 5: **(a) Classical single-root MT, with the traversal path for verification of leaf L_1 highlighted. (b) An SMMT with 2 memory block groups covered by 2 smaller MTs with 2 independent MT roots on the secure processor. In SMMT, the traversal path for L_1 verification has 1 less MT level, due to the smaller individual MTs.**

3.2.1.2 SMMT: Design SMMT partitions the memory into memory block groups (MBGs), assigning an MT to each MBG, and maintaining the corresponding MT roots in an MT-root RAM on the secure processor. Since each memory block group (MBG) comprises a fraction of the entire memory space, The individual MTs spanning each MBG, which is a fraction of the entire memory, are smaller than a single-root MT spanning the entire memory, substantially reducing the MT reads/updates, thereby decreasing (enhancing) NVM energy (system IPC). SMMT does not require any modification to the memory addressing architecture (page tables, TLBs). SMMT implements a simple mechanism to obtain the group index of a memory address (MA), utilized to select the appropriate MT root from the MT-root RAM on secure processor during authentication. For n MBGs, the $\log_2 n$ most significant bits (MSBs) of the physical address provide the group index (G_i), utilized to select the appropriate MT root from the MT-root RAM.

3.2.1.3 SMMT: Overhead Although SMMTs offer significant improvement in system IPC and NVM energy over single-root MT authentication through fewer MT reads and updates, it incurs a high on-chip storage overhead of the MT-root RAM. Whereas the advantages of SMMTs over classical single-root MTs scale logarithmically with the number of MBGs, it comes at the expense of linearly scaling on-chip MT-root RAM.

3.2.2 Dynamic multi-root Merkle Trees (DMMT)

Decentralized dynamic MMTs (DMMTs), as an alternative to SMMTs, provide the NVM energy and system IPC improvements of SMMTs without the significant overhead of processor MT-root RAM. Instead of maintaining individual MTs over all the MBGs like SMMTs, DMMTs maintain a small MT over one frequently accessed MBG, and a larger MT spanning all other MBGs. As discussed below, DMMT uses a low overhead memory access tracking architecture to translate the small MT across MBGs.

3.2.2.1 DMMT: Observation Practical workloads exhibit spatial and temporal locality for memory accesses, i.e., memory accesses are concentrated over a particular MBG (hot MBG henceforth). Therefore, maintaining a smaller hot MT over the hot MBG achieves SMMT-level reduction in the MT read/writes for authentication of a majority of the memory accesses. Since the remaining MBGs (cold MBGs) experience fewer memory accesses, maintaining a larger MT (cold MT henceforth) spanning the cold MBGs requires only one root at the expense of higher MT level traversals for a small fraction of the memory accesses. The DMMT thus stores only two secure roots (hot and cold roots), independent of the number of MBGs.

3.2.2.2 DMMT: Design DMMTs maintain two MTs collectively spanning the memory, a hot MT spanning the hot MBG receiving majority of memory accesses and a cold MT covering the remaining MBGs, with both roots stored on the secure processor. Whereas a memory access to the hot MBG is authenticated with the smaller hot MT terminating at the hot root, an access to any cold MBG is authenticated with the larger cold MT concluding at the cold root.

Figure 6 illustrates DMMT organization. The memory space, T_{MEM}=16, is divided into 4 MBGs, with MBG G_0 designated as the hot MBG and G_1, G_2, and G_3 as the cold MBGs. Please note that the hot and cold MBGs are complementary subsets of the same single-root MT spanning the entire memory space. Figure 6 also highlights the traversed nodes for authenticating (updating) L_1 (black) in the hot MBG, and L_9 (yellow) in the cold MBG. On an L_1 read, the recursive hash and compare procedure of authentication terminates at M_{20}, the hot MT root, which is maintained on the secure processor (R_{HOT}); however, on an L_9 read, the larger cold MT has greater MT traversal levels, concluding at the cold MT root (R_{COLD}). Similarly, on an L_1 write, only M_{10} in the memory

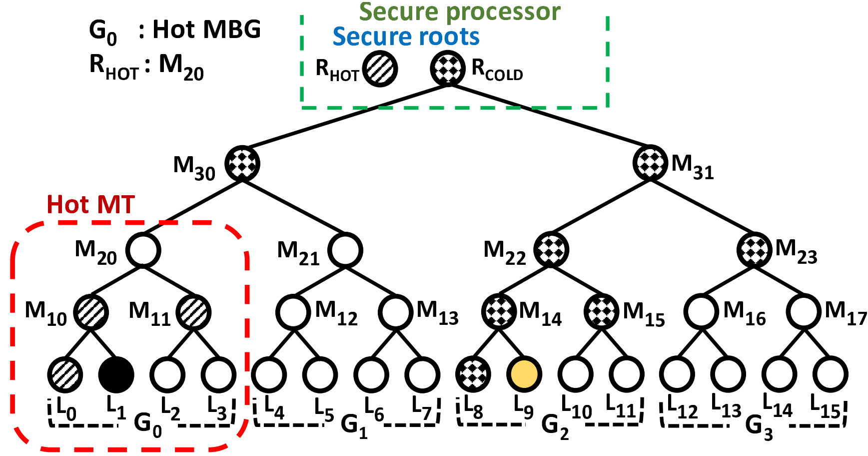

Figure 6: **DMMT organization with the hot MBG spanned by the hot MT, and a cold MT covering all remaining MBGs collectively, with their corresponding roots on the secure processor (R_{HOT} and R_{COLD}). Leaf L_1 (black) in the hot MBG requires lower MT level traversals than L_9 (yellow) in the cold MBG.**

and R_{HOT} are updated. Note that M_{20} and hence M_{30} are not updated during a write to the hot MBG without compromising security. Even though M_{20} is required during authentication of leaf nodes from the cold MBG G_1, the cold MT considers it as an unaltered node. M_{20} tampering can be detected by hashing M_{20} and M_{21}, and comparing it with M_{30}.

3.2.2.3 DMMT: Hot MBG prediction and update Since imprecise hot MBG selection may fail to capture the majority of memory accesses and decrease DMMT efficacy, DMMT requires effective hot MBG prediction for better performance. Also, the hot MBG selection must be dynamic to track the changing patterns of memory accesses, necessitating a simple, dynamic and robust hot MBG prediction architecture for efficient DMMTs. DMMT leverages the spatial and temporal locality of memory accesses in practical workloads for a simple, effective hot MBG prediction architecture. DMMT tracks the memory access count of each MBG over a period of P_{PRED} accesses, and designates the MBG that accounts for the maximum accesses as the hot MBG for the next P_{PRED} accesses. The access count for each MBG is reset after every P_{PRED} accesses for the next prediction cycle. For example, in Fig. 6, G_0 is initially considered the hot MBG, with 0 memory accesses to all the MBGs. Considering P_{PRED}=16, if MBG G_3 receives 10 accesses and G_0, G_1, and G_2 each receives 2 accesses, then DMMT designates G_3 to be the next hot MBG.

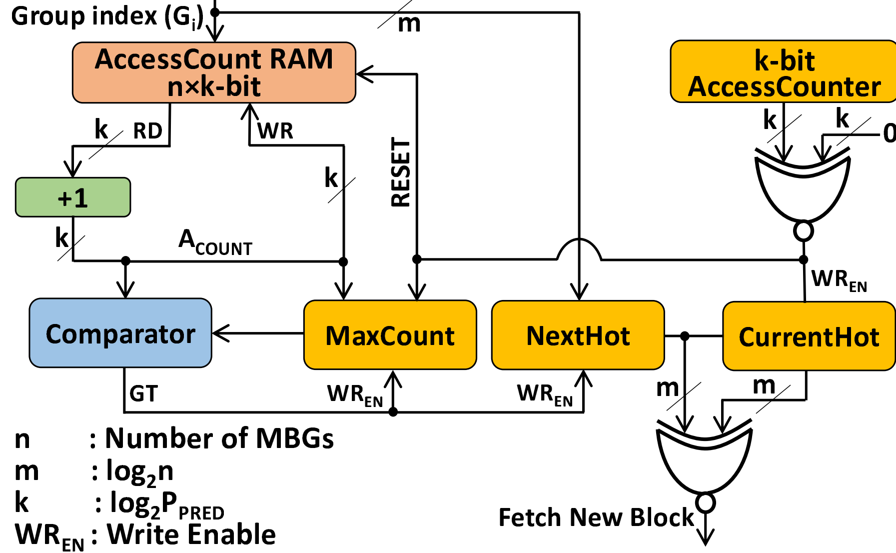

Figure 7: **Hot MBG prediction architecture for DMMT. For the evaluations, n=1024 and P_{PRED}=1024.**

Figure 7 illustrates the hot MBG prediction architecture of DMMT. The *AccessCount* RAM records the access count for each of n MBGs. During a leaf node read/write, the group index (G_{i}) of its corresponding MBG is obtained (refer Sec. 3.2.1), and the RAM content for that G_{i} is incremented by 1. The resultant sum (A_{COUNT}) is compared with the value of *MaxCount* register, which records the maximum access count within P_{PRED} accesses. If A_{COUNT} is greater than *MaxCount*, *MaxCount* is updated with A_{COUNT}, and the corresponding G_{i} is stored in the *NextHot* register that records which MBG has received the maximum accesses within a period. The *AccessCounter* is incremented on every memory access, and reset after P_{PRED} accesses, initiating a new cycle of prediction. When the counter output is 0, the *CurrentHot* register, which records the G_{i} of the current hot MBG, is updated with *NextHot*, and the new hot root is fetched from the memory.

When the predicted hot MBG changes, DMMT updates the old hot root and its corresponding branch in the main memory. Subsequently, the sub-tree root spanning the new hot MBG is fetched, authenticated, and stored as the new hot root (R_{HOT}) on the secure processor. In the example, when the hot MBG changes from G_0 to G_3, the DMMT updates M_{20} with the latest value of R_{HOT}, followed by update of nodes M_{30} and R_{COLD}; M_{23}, the root of the sub-tree covering G_3, is fetched, authenticated, and assigned to R_{HOT}.

3.2.3 ASSURE: Authentication architecture

ASSURE synergistically integrates MMTs with SMACs to extend the NVM energy/lifetime/IPC improvements of SMACs to MMTs. Since SMACs are drop-in replacements of HMACs, ASSURE architects the MMTs with SMAC nodes. ASSURE partitions each SMAC MMT node into k words for k-ary MMTs, and updates only that SMAC word corresponding to the modified child SMAC node. Although data SMACs utilize DEUCE modbits, modbits for counter SMACs in the MMTs are non-existent. ASSURE assigns a modbit to each SMAC node of the counter MMTs to identify its modified/unmodified child nodes, incurring an insignificant memory overhead.

3.2.3.1 Security In ASSURE, SMACs do not alter the cryptographic algorithm of HMACs, maintaining full HMAC entropy; MMTs preserve the hash-and-compare flow with secure root architecture of single-root MTs. Hence, ASSURE preserves the security of the underlying MT memory authentication.

The logic, memory, and latency overhead of ASSURE are evaluated for a typical DEUCE-based encryption architecture with 32-bit line counters. The evaluations consider a 16GB data memory with 1GB counter memory, a 4-ary MMT over the counter memory, and HMAC based on SHA-1 that uses 128-bit codewords [11, 12].

3.2.3.2 Logic overhead The prediction architecture of DMMTs, which dynamically designates the hot MBG, incurs the major logic overhead in ASSURE. To estimate the logic hardware overhead, the prediction architecture designed and synthesized (refer Sec.3.2.2.3) in Verilog, obtaining an estimated overhead of ≈2k 2-input NAND gate count.

3.2.3.3 Memory overhead SMMT requires a $n\times128$-bit MT-root RAM, whereas DMMT requires a $n\times\log_2 P_{\text{PRED}}$ *AccessCount* RAM and a 2-root (hot/cold) RAM, where n is the total number of MBGs and P_{PRED} is the prediction cycle period. DMMT achieves SMMT-level performance with P_{PRED}=1024, for $\approx n\times10$-bit RAM overhead, i.e., 12.8× less overhead than SMMT.

ASSURE also requires modbit storage in main memory for integration of the SMACs with MMTs. ASSURE assigns 1 modbit per 128-bit SMAC MMT node, resulting in (1/128), ≈ 0.78% overhead on the memory allocated to SMAC MMT nodes (ASSURE with SECRET [25] requires additional 1 modbit per 64-bit data word, i.e., ≈ 1.6% memory overhead).

3.2.3.4 Latency overhead The main impact to authentication latency is the reset operation of the *AccessCount* RAM after P_{PRED} memory accesses. Since RAM does not provide RESET ports, it has to be explicitly cleared. For n=1024 and P_{PRED}=1024, DMMT requires 1.25kB RAM, which has an access latency of $\approx$ 1ns, obtained using CACTI 5.3 [79] with low standby power transistors. Therefore, the *AccessCount* RAM reset incurs a latency of $n\times$1ns, i.e., 1024ns, every 1024 memory accesses; this translates to an amortized overhead of 1ns per memory access, which is insignificant compared to high access latencies of NVMs.

3.3 EVALUATION AND RESULTS

ASSURE is evaluated on a TLC RRAM architecture with integer and floating-point workloads from the SPEC CPU2006 [23] benchmarks. A 4-ary MT is considered for the evaluated authentication architectures: BMT [11] authentication (baseline), SMMT ASSURE, and DMMT ASSURE. Without exception, DEUCE is the underlying encryption framework; the use of SECRET, while beneficial to encryption, only marginally improves the results over DEUCE for memory authentication. An MBG count n of 1024, and a prediction period P_{PRED} of 1024 accesses.

3.3.1 Simulation framework

ASSURE is evaluated for NVM energy, lifetime, and system IPC. For NVM energy evaluations, trace-based simulations are performed using NVMain [26]. NVMain is a cycle accurate main memory simulator designed to simulate emerging non-volatile memories at the architectural level. NVMain is configured to reflect a 16GB single channel main memory with 2 ranks and eight x8 devices/rank. The memory controller performs first-ready-first-come-first-serve scheduling, with open page policy. The cell-level energy/latency parameters are provided in [22]. For lifetime evaluation, this dissertation uses an in-house simulator that operates at the page level with a page size of 4kB. Along [22], perfect wear leveling is assumed with a mean cell lifetime of 10^8 writes.

For system IPC evaluations, MARSS [28] is utilized. MARSS is configured to simulate a standard 4-core out-of-order system running at 3GHz. Each core has a private L1 I/D cache of 32kB (latency=2ns) and a private L2 cache of 128kB (latency=5ns). L3 is a shared write-back cache of 8MB (latency=20ns). The 16GB single-channel TLC RRAM main memory has 8 banks;

Table 1: **Workloads comprising of benchmarks from SPEC CPU2006 benchmark suite.**

Workload	Benchmarks	MPKI
WD_1	h264ref, astar, milc, lbm	22.60
WD_2	bzip2, gcc, sphinx3, xalancbmk	21.19
WD_3	perlbench, soplex, bwaves, lbm	20.78
WD_4	bzip2, gcc, mcf, omnetpp	19.09
WD_5	perlbench, leslie3d, GemsFDTD, lbm	16.92
WD_6	h264ref, sjeng, bwaves, povray	16.24
WD_7	bzip2, mcf, namd, omnetpp	15.50
WD_8	perlbench, soplex, milc, povray	13.44

the macro latency parameters are provided in [43]. A 128kB 32-way set-associative counter/MT metadata cache (32kB/core) [12] is integrated inside the memory controller for all the evaluated techniques. The HMAC computation is based on SHA-1 with 80-cycle latency [11, 80].

3.3.2 Workloads

To evaluate system performance, composite memory-intensive SPEC CPU2006 workloads are utilized, with each workload containing 4 benchmarks. Table 1 lists each workload with its constituent benchmarks and memory accesses (measured in L3 misses per kilo-instructions (MPKI)). The simulations run for a representative slice of 1 billion instructions for each workload.

3.3.3 Summary of results

Table 2 summarizes the NVM energy, memory lifetime, and IPC results of baseline (BMT), SMMT ASSURE, and DMMT ASSURE (normalized to baseline BMT). SMMT ASSURE (DMMT ASSURE) reduces NVM energy, and improves memory lifetime/system IPC by 58% (53%), 2.36$\times$ (2.11$\times$)/11% (10%), respectively, over baseline BMT authentication.

Table 2: **Summary of the NVM energy, memory lifetime, and IPC results of baseline (BMT), SMMT ASSURE, and DMMT ASSURE (normalized to baseline BMT).**

Authentication Technique	NVM Energy	Memory Lifetime	System IPC
Baseline (BMT)	1	1	1
SMMT ASSURE	0.41	2.36	1.11
DMMT ASSURE	0.45	2.11	1.10

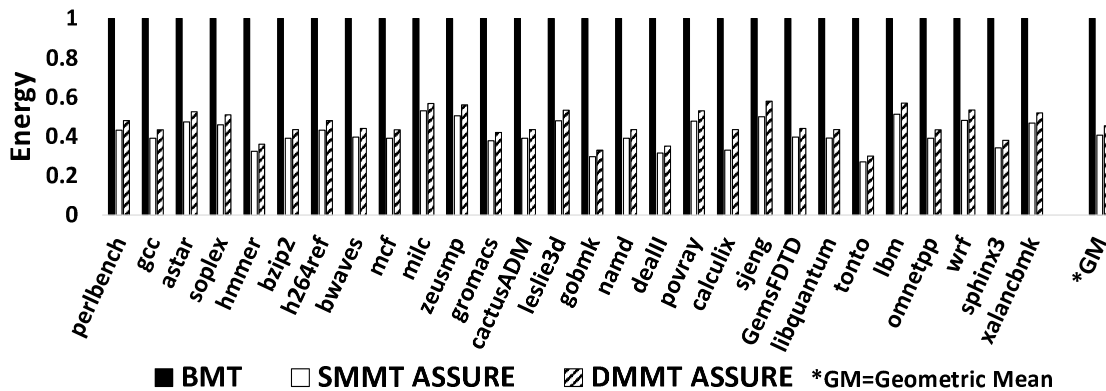

Figure 8: **NVM energy of BMT, SMMT ASSURE, and DMMT ASSURE (normalized to baseline BMT) evaluated on SPEC CPU2006 benchmarks [23] using NVMain [26]. For TLC RRAM, SMMT ASSURE (DMMT ASSURE) reduces NVM energy, on average, by 59% (55%) over baseline BMT authentication.**

3.3.4 NVM energy

Figure 8 illustrates the impact of ASSURE on NVM energy for authentication, with SMMT ASSURE (DMMT ASSURE) reducing NVM energy, on average, by 59% (55%) over BMT authentication. ASSURE leverages the dual advantages of SMACs and MMTs. Whereas SMACs significantly reduce cell writes for data HMACs and each counter MMT node, MMTs decrease the number of MT node reads/writes on each authentication cycle. SMMTs achieve higher energy reduction than DMMTs, because all memory accesses encounter smaller MTs in SMMTs, whereas for DMMTs, memory accesses to the cold MBGs traverse a larger MT with more levels. DMMTs

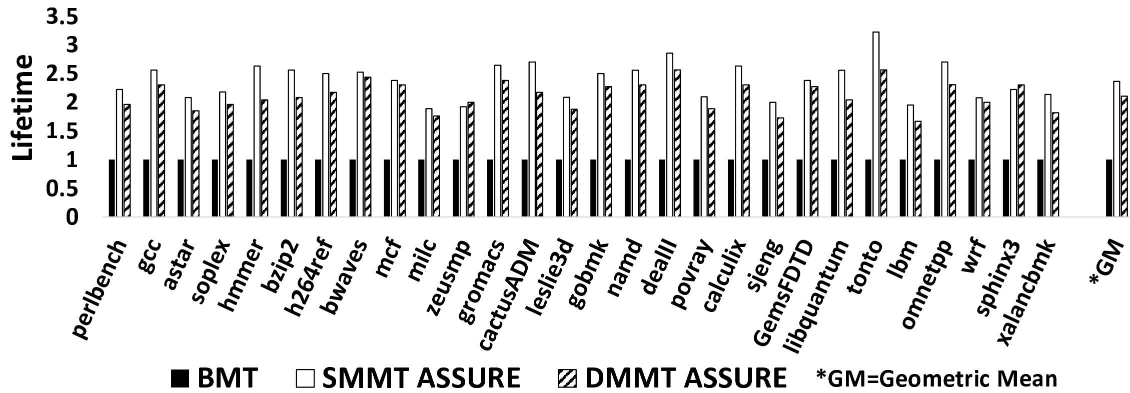

Figure 9: **Memory lifetime of BMT, SMMT ASSURE, and DMMT ASSURE (normalized to baseline BMT) evaluated on SPEC CPU2006 benchmarks [23], utilizing an in-house simulator, using average cell lifetime of 10^8 writes until failure [22]. For TLC RRAM, SMMT ASSURE (DMMT ASSURE) improves memory lifetime, on average, by 2.36$\times$ (2.11$\times$) over baseline BMT authentication.**

achieve $\approx$ 93% of NVM energy reduction capabilities of SMMTs, with $\approx$ 12.8$\times$ smaller RAM overhead (*AccessCount* RAM and a 2-root (hot/cold root) RAM) than the SMMT MT-root RAM.

3.3.5 Memory lifetime

Figure 9 illustrates the memory lifetime improvement offered by SMMT ASSURE (DMMT ASSURE) over baseline BMT. SMMT ASSURE (DMMT ASSURE) extends the memory lifetime, on average, by 2.36$\times$ (2.11$\times$) over baseline BMT, through significant cell write reduction. Cell write reduction results in fewer programmed cells, thereby reducing the wear rate of memory. DMMT ASSURE offers $\approx$ 89% of the lifetime improvement achieved by SMMT ASSURE, with DMMT ASSURE performing marginally worse than SMMT ASSURE due to a small fraction of memory accesses reaching the DMMT cold MBGs.

3.3.6 System performance (IPC)

Figure 10 illustrates the impact of ASSURE on system performance. SMMT ASSURE (DMMT ASSURE) improves the system IPC, on average, by 11% (10%) over baseline BMT. ASSURE implements MMTs that diminish the number of MT node reads/writes by maintaining a smaller

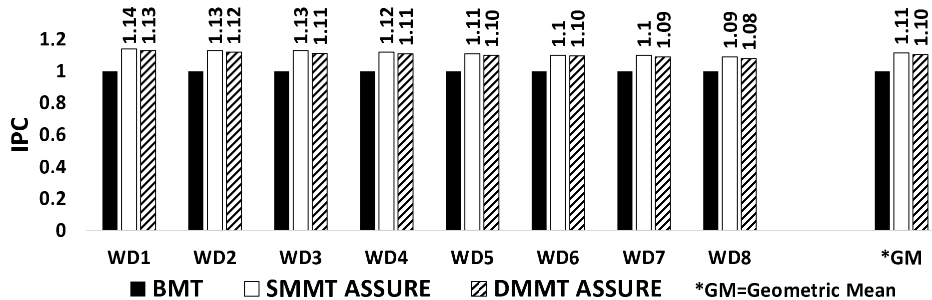

Figure 10: **System IPC of BMT, SMMT ASSURE, and DMMT ASSURE (normalized to baseline BMT) evaluated using workloads comprised of SPEC CPU2006 benchmarks [23] over MARSS [28]. For TLC RRAM, SMMT ASSURE (DMMT ASSURE) improves IPC, on average, by 11% (10%) over baseline BMT authentication.**

MT over the MBGs (hot MBG for DMMTs), thereby reducing bank contention between critical data/counter reads (writes) and MT node reads (writes). NVM systems are power constrained and update only a fixed number of cells per write slot [24]. SMACs in ASSURE enable multiple power-constrained concurrent writes in one write slot by reducing the effective number of cell updates per write, thereby diminishing the effective latency of authentication. The effectiveness of ASSURE becomes more evident for the high MPKI workloads (e.g., WD_1 and WD_2) that require more frequent authentication due to higher memory accesses.

3.3.7 Sensitivity: n and P_{PRED}

Figure 11 illustrates the impact of n and P_{PRED} over the effectiveness of DMMT in terms of average NVM energy, memory lifetime, and IPC, normalized to optimum n and P_{PRED} values of 1024 and 1024, respectively. Higher n values result in MBGs smaller than the spatial locality footprint of the program, suffering higher cold MT reads/writes, which is undesirable. Lower values of n result in larger hot MTs, resulting in increased MT read/writes.

Higher P_{PRED} leads to slower tracking of the memory access pattern change, resulting in higher cold MT reads/writes. Also, lower P_{PRED} values marginally affect DMMT performance for workloads with poor spatial locality, because lower P_{PRED}s lead to frequent updates of the changing hot MT roots and their corresponding branches.

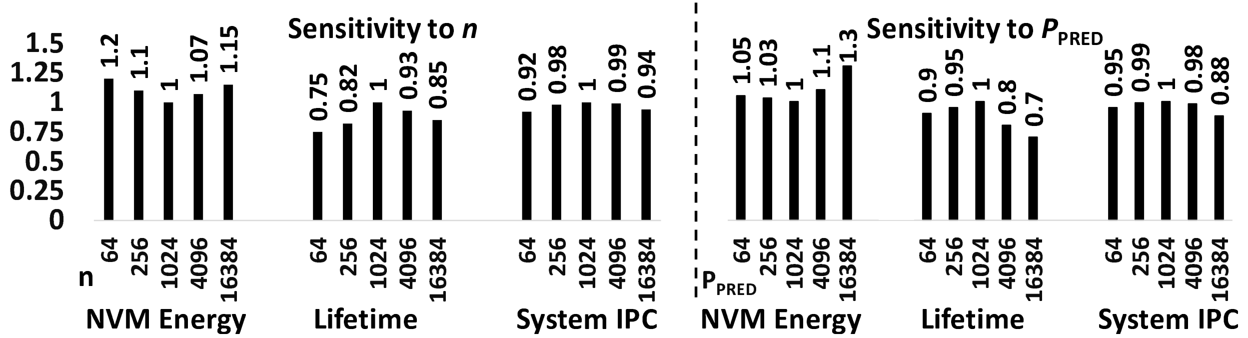

Figure 11: **Sensitivity of NVM energy, lifetime, and system IPC for DMMT to n and P_{PRED}, normalized to n=1024 and P_{PRED}=1024.**

3.4 RELATED WORK

Initial work on efficient MT authentication focused on reducing the significant execution overhead for integrity checks over a complete branch of an MT for a single data cache line. In one of the early works on MT memory authentication, it was shown that the performance overhead can be upto 10× over an unauthenticated memory; to overcome this, MT node caching was proposed as a solution [81]. The primary insight from this work was that cached MT nodes are considered to be trusted, hence, the integrity checking need not proceed till the root, it needs to be executed till the first root on that branch that is found in the cache; caching MT nodes reduced the performance overhead to 25%. In this dissertation, all authentication schemes are evaluated with a per-core MT node cache of 128kB. Further, changing the hash function for MAC generation from SHA-1 to GCM (Galois Counter Mode) allows reduction in hash generation latency [82]; this work also pointed out that the counters need to be protected from replay attacks using MT, ensuring correct encryption and decryption of data. Bonsai Merkle Tree (BMT) presented the most efficient MT architecture, by protecting the counters with an MT, while hashing the data along with MT-protected counters; this prevented replay attacks with a significantly lower overhead because of the smaller MT for counters; this work also demonstrated an efficient architecture to extend the authentication framework to the disk storage [11]. In this dissertation, all evaluated authentication techniques build on the baseline BMT architecture.

In all previous memory authentication scenarios, the operating system (OS) is considered to be a part of the TCB, securing the critical memory management operations like memory alloca-

tion and page table translation. However, this assumption might not hold true in many scenarios, specially considering open-source OS. Malicious OS can subvert the MT built on physical address space using branch splicing attacks, in which the physical address corresponding to a program's virtual address is corrupted, fetching the wrong memory block and also its corresponding wrong MT nodes. However, since the MT nodes fetched correspond to the incorrect data, memory authentication fails to detect this attack. MT authentication over the virtual address space can prevent branch splicing, however, it spans a huge address space of 64 bits, and requires one full tree for each application. Reduced Address Space (RAS) tree resolves this problem by using a dynamic tree that adds pages under the tree as the application footprint increases [83]. However, once an application reaches its maximum footprint, the RAS is not pruned, and is maintained over the complete footprint. Note that a RAS is kept for each running application, so there are multiple root nodes managed on-chip; ASSURE can be integrated seamlessly with RAS to reduce its overheads significantly, especially when the maximum memory footprint is achieved.

3.5 CONCLUSIONS

Memory authentication is key to ensuring data integrity in NVMs. However, in practice, it comes at the expense of increased NVM energy, degraded lifetime, and poor system IPC. ASSURE is the first work to address low cost NVM authentication. ASSURE integrates smart MACs (SMACs) and multi-root MTs (MMTs) to realize tamper-evident NVMs with low energy and improved lifetime as well as IPC. SMACs eliminate redundant HMAC computations of unmodified words on write-backs, reducing cell writes/NVM energy and improving lifetime. MMTs maintain multiple smaller MTs that collectively span the counter memory, reducing MT reads/ writes for authentication, thereby reducing NVM energy, increasing lifetime, and improving system IPC. ASSURE outperforms state-of-the-art NVM authentication with 55% lower NVM energy, 2.11$\times$ improved lifetime, and 10% better system IPC.

4.0 READPRO: READ PROMOTION SCHEDULING IN ORAM

ReadPRO (**Read** **Pro**motion) is a dynamic, read-prioritized ORAM controller scheduling solution to decrease ORAM read latency for efficient ORAM integration in main memories. This chapter provides detailed discussions on the observations, scheduling policy, and architecture of ReadPRO, while also evaluating the advantages of ReadPRO over state-of-the-art ORAM.

4.1 DECOMPOSING ORAM READ LATENCY

On an LLC miss (i.e., when the processor wants to read/write to an LA not present in the LLC), the LLC forwards the LA access to the ORAM controller to fetch the target LA block from main memory. The ORAM controller initially buffers the LA access in the LA queue (refer Fig. 3), and serves them using first-in-first-out (FIFO) scheduling [38, 75]. For example, in Fig. 12, the LA queue receives an LA read, write, and read access from the LLC to LAs 100, 300, and 600, respectively ($LA_{100}/LA_{300}/LA_{600}$ henceforth), in that order. A read/write to LA_i is denoted as R_i/W_i; each R_i or W_i has a corresponding $ORAM_i$, composed of a $ReadPhase_i$ and a $WritePhase_i$. The baseline ORAM first completes $ORAM_{100}$; thereafter, it sequentially completes $ORAM_{300}$ and finally $ORAM_{600}$. The performance-critical LA reads, i.e., R_{100} and R_{600}, receive the data from memory during the read phase operations, i.e., $ReadPhase_{100}$ and $ReadPhase_{600}$, respectively, of their corresponding ORAM accesses.

4.2 INEFFICIENT WRITE BUFFERING IN MEMORY CONTROLLER

Since the LA queue is served in FIFO order, if an LA read access is preceded by LA read or LA write access(es) in the LA queue, the read phase of that LA read access is delayed by the write

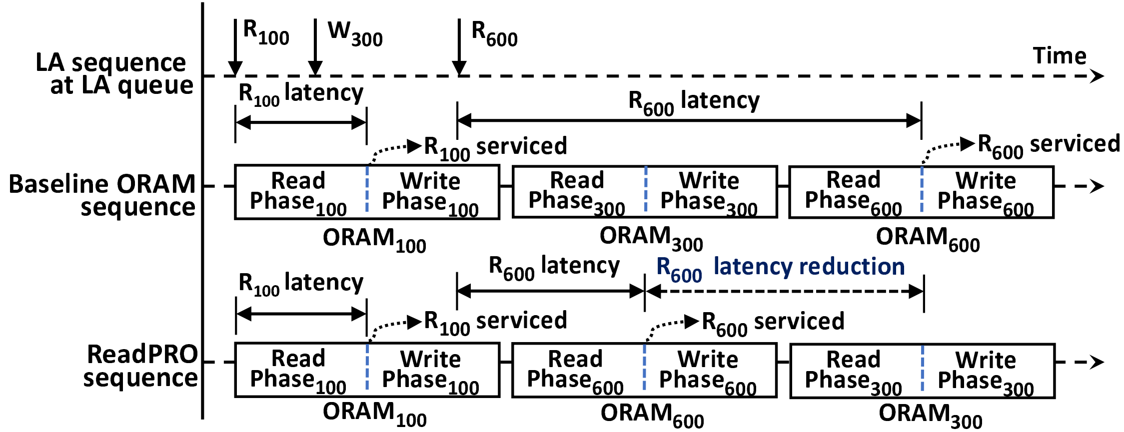

Figure 12: **ReadPRO illustration with a read, write, and read to LA 100, 300, and 600, respectively, in the LA queue. The baseline ORAM serves the accesses in FIFO order, i.e., $ORAM_{100}$, $ORAM_{300}$, and $ORAM_{600}$. In contrast, ReadPRO promotes the $ORAM_{600}$ over $ORAM_{300}$, reducing the latency of critical-path-bound R_{600} significantly.**

phase operations of preceding LA read accesses, and both read and write phase operations of the preceding LA write accesses, all of which are not on the critical path of program execution. For example, in Fig. 12, R_{100} and W_{300} precedes R_{600} in the LA queue; therefore, $ReadPhase_{600}$ incurs the additional latency of $WritePhase_{100}$, $ReadPhase_{300}$ and $WritePhase_{300}$.

Traditionally, the modern memory controller on the processor prioritizes memory reads by maintaining a write buffer [84, 85]; the write phase operations of older LA read and write accesses that delay the read phase of a new LA read can be cached by the write buffer, allowing progress of the succeeding read phase for the new LA read. However, when the write buffer is full (or reaches a high watermark), the writes in the buffer are drained till the buffer is empty (or attains a low watermark). The following observations are on the effectiveness of write-buffer-based read-prioritization in the memory controller *after* LA access translation into its read and write phase by the address logic in the ORAM controller. (i) *Although an infinite capacity write buffer will cache the write phase operations of all ORAM accesses, the read phase operations of the older LA writes (e.g., $ReadPhase_{300}$), which is not on the critical path, will still be present and delay the critical-path-bound read phase of a newer LA read access (e.g., $ReadPhase_{600}$)*. (ii) *Since each LA read/write access spawns multiple memory writes in the write phase, a practically-sized write buffer will overflow within a few LA accesses, offsetting the read-prioritization advantages of write buffers in memory controllers*. For example, consider a 4GB ORAM, similar to [39], with Z=4 and

block size of 64 bytes, realizing an ORAM tree of 24 levels. With tree-top caching of the top 10 levels in a 256kB ORAM cache [39], the main memory stores the remaining 14 levels. During an ORAM access, the memory controller fetches and writes back 4 data blocks and 1 meta-data block per bucket, i.e., a total of 14*5=70 blocks. A reasonably sized 64-entry write buffer [84, 86] will overflow on every write phase and drain the block writes to the current path.

Previous work on write traffic reduction in ORAM can effectively achieve improvements over the baseline ORAM by reducing the average number of block writes on an ORAM access, delaying write buffer overflow. Fork Path [38] identifies the overlapping sections of path between two consecutive memory accesses and prevents the write back of blocks in the overlapping parts of paths during the consecutive ORAM accesses. However, [39] demonstrates that with a reasonable tree-top caching, very few consecutive ORAM accesses overlap in memory; although tree-top caching is considered in this example, it does not prevent frequent overflow of a reasonably sized write buffer. RAW ORAM (a.k.a Tiny ORAM) [41] executes two different and periodically interleaved ORAM accesses; access-only (AO) and eviction-only (EO) accesses. AO accesses serve LA accesses, and entails reading of all blocks on a path, and write-back of only the metadata blocks. In contrast, EO accesses evict data blocks from stash, wherein all blocks are read and written back from a randomly selected path; an EO is performed periodically after a fixed number (A) of AO accesses (generally, A=5). So, following the example, on an AO access, 14 blocks are written back, and after 5 AO accesses (i.e., 14*5=70 block writes), the write buffer overflows.

ReadPRO effectively addresses (i) the issue of read access read phase operations incurring additional latency of the read and write phase operations of older LA writes, and (ii) inefficient read-prioritization by the memory controller write buffer. ReadPRO is motivated by the key observation that *promoting newer LA reads over older LA writes at the LLC-ORAM interface advances the read phase of the LA reads, ensuring faster critical data fetch without being delayed by the non-critical read and write phase operations of older LA writes*. Figure 12 illustrates this observation motivating ReadPRO: Prioritizing the newer R_{600} over older W_{300} promotes $ORAM_{600}$ over $ORAM_{300}$, advancing $ReadPhase_{600}$ ahead of the $ReadPhase_{300}$ **and** $WritePhase_{300}$ thereby reducing the effective latency of the performance-critical R_{600}. Note that $ReadPhase_{600}$ is still delayed by $WritePhase_{100}$ which must follow $ReadPhase_{100}$ before initiating successive ORAM accesses to preserve security [38, 39, 87].

4.3 READPRO ARCHITECTURE

Figure 13 illustrates the primary components of the ReadPRO scheduler architecture: (i) independent queues for LA reads (LA_{rd}) and writes (LA_{wr}), (ii) LA access checker, (iii) ReadPRO arbiter, and (iv) logic for dynamic ReadPRO threshold (Th_{PRO}) enforcement. ReadPRO replaces the single LA queue with an LA_{rd} *queue* to buffer LA reads, and an LA_{wr} *queue* to buffer LA writes. On an LA access due to an LLC miss, the *LA access checker* determines whether the access is an LA read or LA write, and then forwards it to the LA_{rd} queue or LA_{wr} queue, respectively. The LA access checker also provides a unique time-stamp to each queued access by maintaining a 64-bit counter (reset on reboot) that is incremented with every LA access arrival; the counter values are stored with the LA accesses in their respective queues to facilitate scheduling. Individually, the LA_{rd} queue and the LA_{wr} queue are served in FIFO order.

The *ReadPRO arbiter* schedules an LA access for ORAM access from the LA_{rd} or the LA_{wr} queue, based on their head (oldest) entries; if either queue is empty, entries from the non-empty queue are scheduled for ORAM access. Otherwise, if the counter value of the LA_{rd} head is less than the counter value of the LA_{wr} head, i.e., if the LA_{rd} head is older than the LA_{wr} head, the LA_{rd} head is scheduled for ORAM access. Additionally, even if the LA_{wr} head entry is older than the LA_{rd} head, ReadPRO schedules the newer LA_{rd} head for ORAM access, promoting it over the older LA write. Ideally, ReadPRO will always maintain the priority, LA read $\prec$ LA write. Therefore, ReadPRO continues promoting newer LA reads over older LA writes; however, considering a finite capacity LA_{wr} queue, if more than Th_{PRO} LA reads have been promoted over the oldest LA write, ReadPRO is suspended, and the LA_{wr} head entry is scheduled for ORAM access. Th_{PRO} is thus a threshold that ensures that the promotion of LA reads does not result in the starvation of LA writes. Without such a throttling provision, the LA_{wr} queue will overflow frequently to drain the LA writes, stalling LA reads in the LA_{rd} queue for long intervals.

Dynamic threshold of buffering: The threshold value Th_{PRO} is adjusted dynamically based on the ratio of the arrival rate of the LA reads and writes. ReadPRO starts with a base Th_{PRO} value of Th^{base}_{PRO}, and updates the value periodically after every T_{EPOCH} LA accesses issued by the LLC. ReadPRO determines the number of LA reads (N_{rd}) and LA writes (N_{wr}) within a T_{EPOCH}. The new Th_{PRO} for the next T_{EPOCH} LA accesses is Th^{new}_{PRO}=$Th^{base}_{PRO}*(N_{rd}/N_{wr})$. When LA reads are

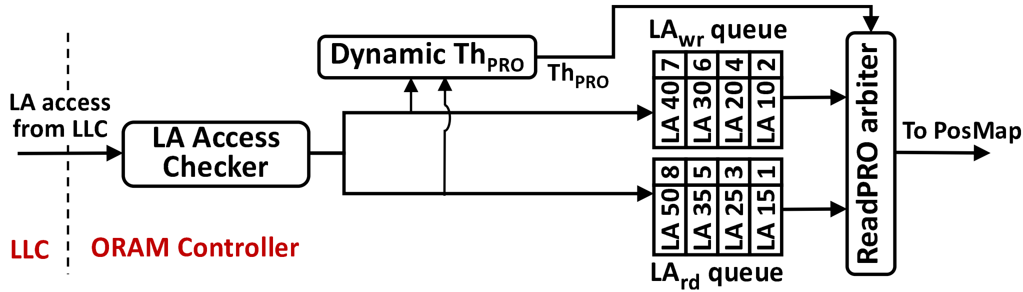

Figure 13: **ReadPRO architecture: (i) The LA access checker receives accesses from LLC, determines if it is a read or write, and forwards them to LA_{rd} or LA_{wr} queue, respectively, with the time-stamp. (ii) The dynamic Th_{PRO} evaluator updates Th_{PRO} at runtime after T_{EPOCH} LA accesses. (iii) The ReadPRO arbiter logic schedules an LA access from the LA queues following read promotion, while bound by Th_{PRO}.**

more frequent than LA writes, i.e., $N_{\mathrm{rd}}>N_{\mathrm{wr}}$, $\mathrm{Th}_{\mathrm{PRO}}^{\mathrm{new}}$ increases, favoring an increase in LA read promotions; a higher frequency of LA reads facilitates longer buffering of LA writes in the $\mathrm{LA}_{\mathrm{wr}}$ queue, without increasing the probability of queue overflow. In contrast, for more frequent LA writes, $N_{\mathrm{rd}}/N_{\mathrm{wr}}$ and subsequently $\mathrm{Th}_{\mathrm{PRO}}^{\mathrm{new}}$ decreases, reducing the number of LA read promotions, thereby decreasing the $\mathrm{LA}_{\mathrm{wr}}$ queue overflow probability.

Data dependency while scheduling: Although ReadPRO alters the ordering of memory accesses, it does not introduce data dependency hazards. ReadPRO only promotes LA reads over LA writes; therefore, ReadPRO does not introduce read-after-read, write-after-read, and write-after-write data hazards by construction. The only concern is the read-after-write (RAW) dependency, wherein a newer LA read is promoted over an older LA write to the same address. In ReadPRO, before an LA read is pushed into the $\mathrm{LA}_{\mathrm{rd}}$ queue, the $\mathrm{LA}_{\mathrm{wr}}$ queue is scanned in hardware from the tail (most recent entry) to the head for an older write to the same LA. If present, the LA read is returned with the most recent write data, and is not pushed into the $\mathrm{LA}_{\mathrm{rd}}$ queue; therefore, the LA queues in ReadPRO do not contain a read and an older write to the same LA, eliminating RAW hazards. The $\mathrm{LA}_{\mathrm{wr}}$ queue scan can be a concurrent search supported by a CAM, or a sequential search supported by the nominal queue read/write circuit. Although CAM search has low latency, the CAM circuit has high area and power overheads; therefore, ReadPRO employs a sequential search for low area and power overhead, but higher latency. However, the sequential queue scan is faster than an ORAM access; hence, the scan latency is hidden by overlapping it with the ORAM access.

Hardware overhead: ReadPRO incurs (i) logic overhead for the LA access checker, the ReadPRO arbiter, and dynamic Th_{PRO} evaluator, along with (ii) memory overhead for time-stamp counters in the LA_{rd} and LA_{wr} queues. The LA access checker, ReadPRO arbiter, and dynamic Th_{PRO} evaluator modules are designed and synthesized in Verilog for an estimated logic overhead of $\approx$ 5k 2-input nand gates. Although ReadPRO replaces the single LA queue in baseline ORAM with two independent LA queues, the cumulative queue capacity is kept similar to the single LA queue in baseline ORAM; for an N-entry LA queue in baseline ORAM, ReadPRO maintains $N/2$-entry LA_{rd} and LA_{wr} queues. The on-chip memory overhead for the time-stamp counters in the LA_{rd} and LA_{wr} queues is N*64 bits; for e.g., given 64-entry LA_{rd} and LA_{wr} queues, the memory overhead is 64*2*8 bytes, i.e., 1kB.

4.4 SECURITY OF READPRO

The baseline ORAM prevents information leakage about the LA accessed, memory access type, and associated data, while ensuring negligible stash overflow probability to preserve security [34]. ReadPRO provides security guarantees equivalent to baseline ORAM, i.e., prevents information leakage about the data, type, or address of memory access. ReadPRO adopts measures identical to the baseline ORAM to obfuscate the plaintext data, the memory access type, and the accessed LA. ReadPRO keeps data encrypted, performs a read and a write phase on every ORAM access, and utilizes a secure random leaf label remapping for LAs, identical to the baseline ORAM. ReadPRO implements the same stash eviction algorithm with identical stash size and number of ORAM levels as the baseline ORAM; hence, it is guaranteed to keep the probability of stash overflow unaffected. Although ReadPRO reorders LA accesses based on the type (read/write) of the accesses, it preserves security since the adversary still observes independent path accesses in the memory due to random leaf remapping equivalent to the baseline ORAM. The path access sequence is different even for 2 consecutive runs of the same application due to the random leaf re-mapping; therefore, the attacker cannot infer the re-ordered access by comparing access patterns from 2 different runs of the same application. Additionally, previous work in Fork Path [38] has shown that LA access re-ordering is secure.

Table 3: **Configuration for evaluation setup**

MARSS		DRAMSim2	
Core	4 out-of-order, 2GHz	Memory size	16GB, 1.33 GHz
Private L1 (I/D) cache	32kB, 2-way, 2ns	Channel; rank; bank	2 channels; 2 ranks/ch; 8 x8 banks
Private L2 cache	128kB, 8-way, 5ns	Scheduling	First ready-first come-first serve
Shared L3 cache	4MB, 32-way, 20ns	Row buffer management	Open page policy
Cache line size	64 bytes	DRAM write-buffer size	128 entries (64 entries/channel)

4.5 EVALUATION AND RESULTS

4.5.1 Methodology

ReadPRO is compared to a baseline RAW ORAM [41] implementation; RAW ORAM has lower write traffic and better performance in comparison to nominal Path ORAM. ReadPRO is evaluated on a DDR3 DRAM architecture with the SPEC CPU2006 benchmark suite [23]. Full-system simulations are utilized to evaluate average read latency and system IPC, using MARSS [28] coupled with DRAMSim2 [88] for monolithic, cycle-accurate system simulation with a detailed memory model. For multi-core simulations, identical benchmarks are executed on each simulated core. The simulation configuration is summarized in Table 4 and the DDR3 timing parameters are obtained from [89]; $\mathrm{Th}_{\mathrm{PRO}}^{\mathrm{base}}$=5 and T_{EPOCH}=128.

The parameters of the evaluated ORAM architectures are representative of the optimal configurations obtained through design exploration in [41] and used widely in [35, 38, 39, 87]. An 8GB ORAM with Z=4 and stash size of 200 is considered for evaluations. Both the baseline and ReadPRO assume 50% utilization, i.e., upto 50% of the main memory is assumed to hold data blocks. Therefore, an 8GB ORAM requires 16GB DRAM resulting in an ORAM tree of 26 levels. Both the baseline and ReadPRO implement tree-top caching [39] of 1MB, effectively storing 12 levels from the root on-chip in the TCB. The address mapping scheme of "row:bank:column:rank:channel" with a sub-tree layout approach is also adopted from [87], while both baseline and ReadPRO integrate PrORAM prefetching [35] and LEO write optimization [40].

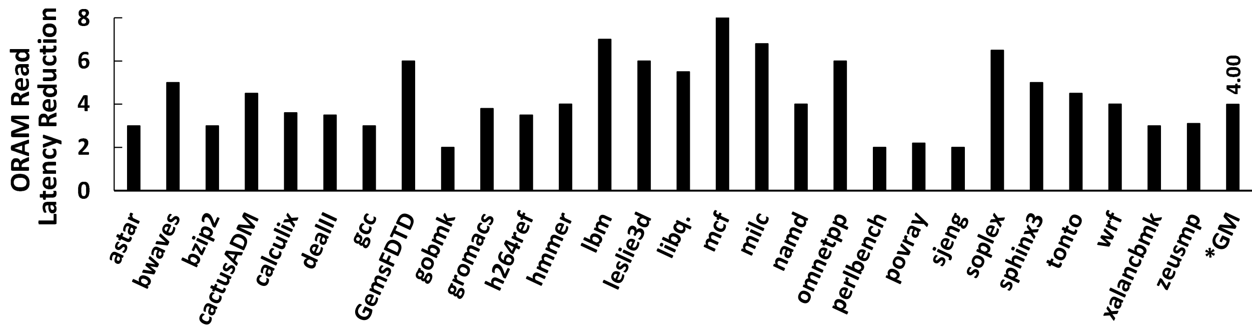

Figure 14: **ORAM read latency reduction of ReadPRO (normalized to the baseline) evaluated on SPEC CPU2006 benchmarks [23] using MARSSx86 [28] coupled with DRAMSim2 [88]. For a DDR3-1333 memory system, ReadPRO reduces average ORAM read latency by 4× over the baseline.**

4.5.2 ORAM read latency

As shown in Fig. 14 ReadPRO reduces the average effective ORAM read latency by 4× over the baseline. ReadPRO conditionally promotes LA reads over LA writes, prioritizing the critical-path bound read access read phase operations to return data faster to the processor. The benchmarks with a high LA access rate (measured in reads-per-kilo-instruction (RPKI)) coupled with a high cumulative memory access rate (misses-per-kilo-instruction (MPKI)), like lbm, mcf, milc, and soplex, demonstrate higher improvements in ORAM read latency. A higher RPKI along with high MPKI leads to faster accumulation of LA accesses in the LA queue; however, the ORAM access latency is almost constant across different applications. Hence, LA reads from benchmarks with high MPKI wait for a longer time in the LA queue of the baseline ORAM, obstructed by LA writes. ReadPRO splits the LA queue and prioritizes LA reads, resulting in faster LA_{rd} queue service.

4.5.3 Speedup

As shown in Fig. 15, ReadPRO improves the average system IPC by 38% over the baseline. Since LA reads are on the critical path of program execution, reducing ORAM read latency accelerates program execution, thereby improving system performance. System IPC of benchmarks with high RPKI is more dependent on ORAM read latency in comparison to low RPKI benchmarks; due to high RPKI, more instructions are stalled, waiting for data from memory. As a result, ReadPRO shows improved speedup on high RPKI benchmarks like lbm, mcf, milc, and soplex.

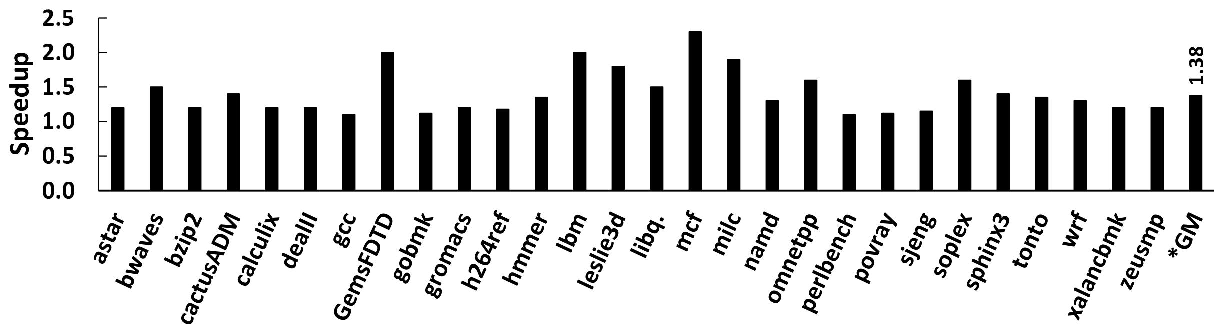

Figure 15: **System performance improvement of ReadPRO (normalized to the baseline) evaluated on SPEC CPU2006 benchmarks [23] using MARSSx86 [28] coupled with DRAM-Sim2 [88]. For a DDR3-1333 memory system, ReadPRO improves system performance by 38% over the baseline.**

4.5.4 Effect of memory controller write buffer size

Figure 16 illustrates the effect of increasing write buffer size of the on-chip memory controller in the baseline, while keeping write buffer size in ReadPRO constant (64 entries/channel). With a write buffer size of 64/256/1024 entries per channel, ReadPRO reduces the average ORAM read latency by 4×/3.8×/3.3×, while increasing system IPC by 38%/36%/32%. Therefore, even with a 16× increase in write buffer size in the baseline, ReadPRO provides substantial improvements in ORAM read latency and system IPC. A higher capacity write buffer integrated with the on-chip memory controller (MC) can cache more writes of the write phase operations; therefore, increasing the MC write buffer in baseline ORAM reduces baseline ORAM read latency and improves system IPC. In comparison, it can be observed that ReadPRO is more efficient than scaling the MC write buffer. ReadPRO operates at the LLC-ORAM interface and caches untranslated LA write accesses, whereas the MC write buffer caches the write phase after ORAM address translation; since a single ORAM access results in multiple memory write operations, it increases the overflow probability of the MC write buffer, reducing its efficiency in read prioritization.

4.6 RELATED WORK

Among architecture-based solutions to improve ORAM efficiency, read latency improvement techniques are most relevant to this work. Co-operative Path ORAM (CP-ORAM) [39] prevents bandwidth wastage in secure application (S-app) and non-secure application (NS-app) co-run scenarios,

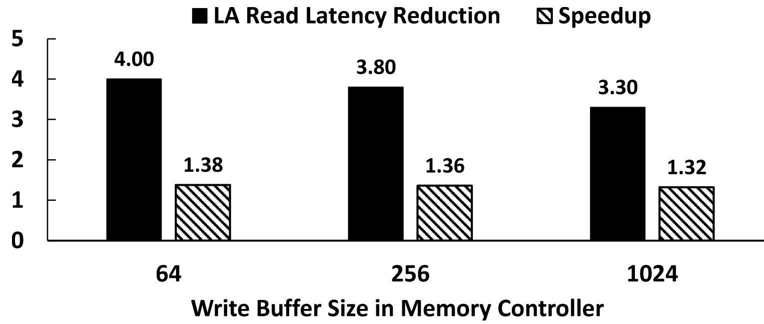

Figure 16: **Effects of increasing write buffer size in memory controller of baseline ORAM on ReadPRO results; for a write buffer size of 64/256/1024, the ORAM read latency reduction is 4×/3.8×/3.3× and speedup is 38%/36%/32%.**

resulting in memory traffic interference of S- and NS-app, leading to channel bandwidth wastage; CP-ORAM utilizes this wasted bandwidth to prefetch blocks for read phase of the succeeding ORAM access. This chapter considers the scenario where only S-apps run, without bandwidth wastage and interference from NS-apps. Additionally, ReadPRO is orthogonal to and easily integrable with CP-ORAM. PrORAM [35] is another ORAM prefetching scheme that maps blocks with high spatial locality to the same path and prefetches them on a single path read; PrORAM is integrated in both baseline and ReadPRO. Recent work on efficient ORAM has leveraged secure-chip on DIMM architectures to offload the ORAM controller to the DIMM, reducing bandwidth burden on processor-memory channels and increasing parallelism in ORAM accesses [3,4]. However, [3,4] assume a more relaxed TCB than ReadPRO, wherein part of the DIMM, i.e., the ORAM controller is secure. Additionally, since these architectures follow the Path ORAM protocol at their core, ReadPRO is easily integrable with the on-DIMM ORAM controller in [3,4] to reduce effective LA read latency and improve their performance.

4.7 CONCLUSIONS

ReadPRO is an architecture for cost-effective integration of ORAM in main memories to thwart access-pattern-based confidentiality attacks. ReadPRO improves the ORAM read latency by conditionally prioritizing critical-path bound LA reads over LA writes in ORAM access scheduling, enhancing overall system performance. ReadPRO splits the single LA queue in the LLC-ORAM

interface into two separate queues for LA reads and LA writes. ReadPRO temporally promotes newer LA reads over older LA writes, eliminating the addition of read and write phase latencies of older LA writes to LA read latency, enabling faster performance-critical data fetches.

5.0 LEO: LOW OVERHEAD ENCRYPTION ORAM

This chapter provides a detailed discussion on LEO, a low penalty design for encryption in ORAM for NVMs. LEO is a secure, optimized two-level counter mode encryption (CME) architecture that reduces the redundant re-encryption of unmodified data or dummy blocks during the write phase of an ORAM access, decreasing the associated cell write overhead.

5.1 WRITE PHASE AND ENCRYPTION

To understand the core architecture of LEO, it is first essential to understand the mechanism of the write phase of an ORAM access, the associated encryption framework in the baseline ORAM, and its security guarantees. In the baseline ORAM, all possible blocks are evicted from the stash during the write phase of an ORAM access to ensure low stash occupancy and negligible stash overflow probability. For a block to be evicted to path-l, it can either be mapped to leaf l, or a leaf whose corresponding path intersects and shares bucket(s) with path-l; it is evicted to a dummy block in the common bucket closest to the leaf level [34, 35, 37, 39, 87]. During eviction, ORAM employs CME to encrypt the blocks of a bucket before they are written back to the NVM [87]. To support CME, each ORAM bucket stores a 64-bit bucket counter (BuCtr) that is incremented whenever one of its constituent blocks is written on a path access. The plaintext in the bucket is partitioned into chunks of AES block size (usually 128 bits [5]), and are encrypted by XORing with a CME one-time pad (OTP) given by $AES_K(\text{BuCtr}||\text{BuID}||\text{ChkID})$, where K is the secret AES key, $||$ is the concatenation operator, BuID, i.e., bucket ID is the unique identifier for each bucket, and ChkID, i.e., chunk ID is the unique identifier of the chunk in the bucket [87]. Bucket IDs start with 1 for the root bucket, then 2 and 3 for its left and right child, and so on.

The write phase in an ORAM access triggers the re-encryption of all blocks in each bucket on the accessed path. The rationale behind re-encrypting all blocks in each bucket is two-fold: (i) re-

encrypting all Z blocks within a bucket prevents the attacker from differentiating between data and dummy blocks [34], and (ii) blocks in every bucket are re-encrypted to prevent identification of the bucket containing the desired LA block or newly evicted blocks [87]. Moreover, the BuCtr is incremented on modification of any block inside the bucket, avoiding counter and OTP reuse for different data [87]. Since the BuCtr is shared by the constituent blocks, incrementing BuCtr changes the OTP, requiring a mandatory re-encryption of every block inside the bucket, regardless of its status (modified/unmodified).

Although all blocks are re-encrypted upon eviction from the stash, in practice, there are three different categories of blocks written back to the path: (i) blocks with unmodified data returned to the same buckets from which they were fetched in the preceding read phase (termed as B_{SAME} blocks), (ii) blocks previously present on the path but returned back to a different bucket, or to the same bucket with modified data (B_{DIFF} blocks) and (iii) blocks not present previously that are newly evicted to replace a dummy block on the path ($B_{\text{NEW_EV}}$ blocks).

5.2 LEO DESIGN

5.2.1 Observation

In the write phase, the majority of blocks written back to a path from the stash are B_{SAME} blocks; only a few are $B_{\text{NEW_EV}}$ or B_{DIFF} blocks. Note that in a path access, only one modified data block (requested for an LA write) can return to its previous bucket. During the write phase, the $B_{\text{NEW_EV}}$ and B_{DIFF} blocks represent modified data in the buckets that are mandatorily re-encrypted with new OTPs to prevent OTP reuse in CME [24,87]. Although the B_{SAME} and dummy blocks are also re-encrypted, it is not required by CME since the data is unmodified [24,25]; this re-encryption is mandated by the ORAM algorithm for access pattern obfuscation.

5.2.2 Design

LEO primarily reduces NVM writes by decreasing the re-encryption of unmodified B_{SAME} and dummy blocks during the ORAM write phase, without affecting the security guarantees of the baseline ORAM. LEO defines the concept of *mandatory re-encryptions* (MR) to refer to the

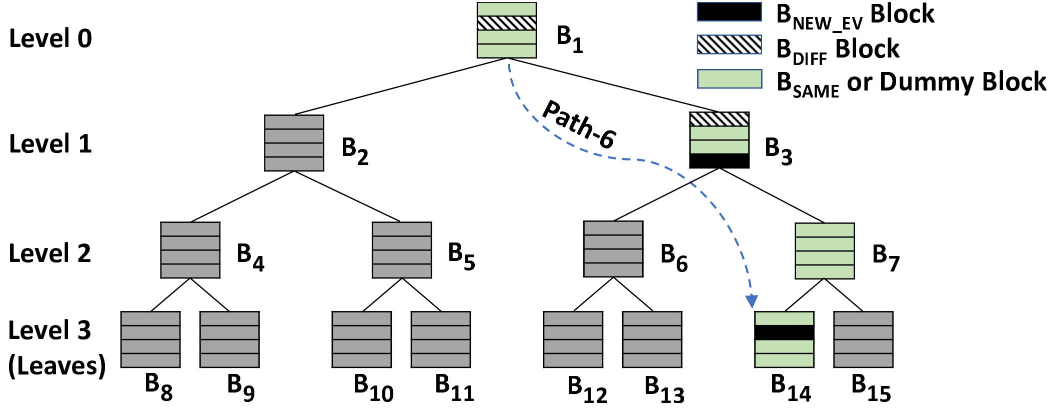

Figure 17: **LEO illustration with L and Z values of 3 and 4, respectively, and path-6 being written back. The $\mathrm{MR_{COUNT}}$ of buckets B_1, B_3, B_7, and B_{14} are 1, 2, 0, and 1, respectively, resulting in $\mathrm{MR_{MAX}}$ of 2. LEO randomly selects 1, 0, 2, and 1 B_{SAME}/dummy blocks from buckets B_1, B_3, B_7, and B_{14}, respectively, for $\mathrm{MR_{MAX}}$ total re-encryptions in each bucket. Here, LEO reduces $[(L+1)\times(Z\text{-}\mathrm{MR_{MAX}})]$, or [4*(4-2)]=8 cache line writes.**

security-critical re-encryptions of $B_{\mathrm{NEW_EV}}$ and B_{DIFF} blocks required by CME. This is because $B_{\mathrm{NEW_EV}}$ and B_{DIFF} blocks hold modified data, requiring re-encryption with a new OTP to prevent OTP reuse. The $\mathrm{MR_{COUNT}}$ of a bucket refers to the total number of $B_{\mathrm{NEW_EV}}$ and B_{DIFF} blocks evicted to that bucket, resulting in as many MRs. For example, in Fig. 17, the $\mathrm{MR_{COUNT}}$ of buckets B_1, B_3, B_7, and B_{14} on path-6 are 1, 2, 0, and 1, respectively. The $\mathrm{MR_{MAX}}$ of a path denotes the highest $\mathrm{MR_{COUNT}}$ over all buckets on that path during a write phase. In Fig. 17, $\mathrm{MR_{MAX}}$ for path-6 is 2. Note that the $\mathrm{MR_{COUNT}}$ of buckets on a path and the path $\mathrm{MR_{MAX}}$ are evaluated dynamically on every path write and are not stored in the NVM.

LEO is designed such that all the buckets on a path enforce $\mathrm{MR_{MAX}}$ (and not Z) block re-encryptions during the write phase of an ORAM access, with mandatory re-encryptions of the $B_{\mathrm{NEW_EV}}$ and B_{DIFF} blocks of a bucket. Note that if the $\mathrm{MR_{COUNT}}$ of a bucket is less than $\mathrm{MR_{MAX}}$, then ($\mathrm{MR_{MAX}}$-$\mathrm{MR_{COUNT}}$) random B_{SAME} or dummy blocks in that bucket are also re-encrypted, for a total of $\mathrm{MR_{MAX}}$ re-encryptions; the remaining blocks are not re-encrypted. Intuitively, if $\mathrm{MR_{MAX}}$ for a path is Z, LEO is equivalent to the baseline ORAM. For example, in Fig. 17, $\mathrm{MR_{MAX}}$ of path-6 is 2 and $\mathrm{MR_{COUNT}}$ of bucket B_1 is 1, hence, one block is chosen randomly from B_1's B_{SAME}/dummy blocks for a total of $\mathrm{MR_{MAX}}$ block re-encryptions in B_1. Similarly, for bucket B_7, two B_{SAME}/dummy blocks are selected randomly for re-encryption. Therefore, LEO decreases

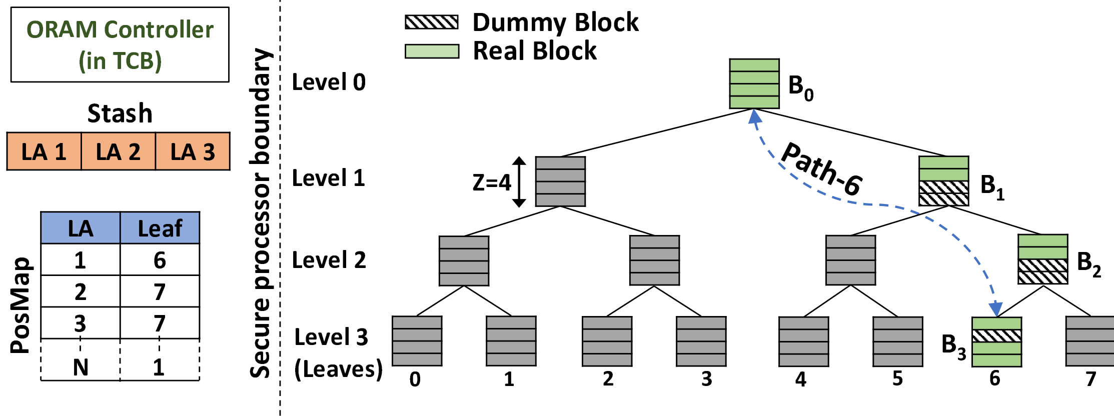

Figure 18: **The internal states of the Path ORAM tree and the ORAM controller prior to the path-6 write. Figure 19 and Figure 20 contrasts the path write scenario for DEUCE-based ORAM against LEO.**

(Z-$\mathrm{MR}_{\mathrm{MAX}}$) block re-encryptions and associated NVM writes for each bucket on a path. For an ORAM with L+1 levels, LEO reduces $[(L+1)\times(Z\text{-}\mathrm{MR}_{\mathrm{MAX}})]$ block writes in the write phase of a single ORAM access.

If the ORAM is accessed when the stash is empty (i.e., no $B_{\mathrm{NEW_EV}}$ blocks), and if no pre-existing blocks are pushed down deeper towards the leaves (i.e., if there are no B_{DIFF} blocks), then the mandatory re-encryption count ($\mathrm{MR}_{\mathrm{COUNT}}$) of all the buckets is 0, resulting in an $\mathrm{MR}_{\mathrm{MAX}}$ of 0. This will reveal to the attacker that no real block was evicted on the path write. To address this corner-case security vulnerability and prevent information leakage when $\mathrm{MR}_{\mathrm{MAX}}$ is 0, LEO adds a security-preserving feature: If the $\mathrm{MR}_{\mathrm{MAX}}$ on a path write is 0, LEO assigns a random $\mathrm{MR}_{\mathrm{MAX}}$ to the path in the range $1 \leq \mathrm{MR}_{\mathrm{MAX}} \leq Z$ determined on the secure processor in the TCB. Therefore, a path write with $\mathrm{MR}_{\mathrm{MAX}}$ value of 0 is rendered indistinguishable from an ORAM access when $\mathrm{MR}_{\mathrm{MAX}}$ is not 0. This ensures that $\mathrm{MR}_{\mathrm{MAX}}$ of a path write is never 0, preventing information leakage in the absence of $B_{\mathrm{NEW_EV}}$ and B_{DIFF} blocks.

5.2.3 Discussion: Partial line encryption vs LEO

Partial line encryption (PLE) or encrypting only the modified blocks have already been proposed in [24, 25, 70, 90] for efficient NVM encryption. The primary objective of PLE is to prevent re-encryption of the unmodified words in a NVM cache line during a cache line write, reducing

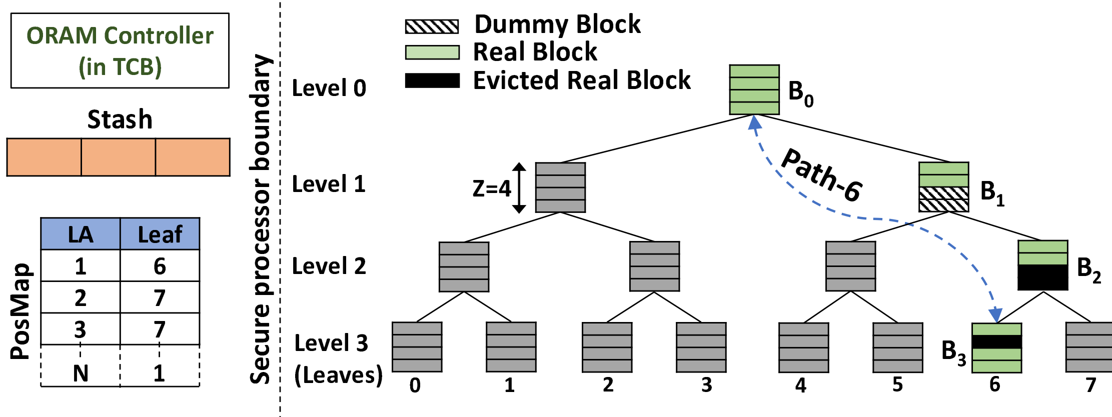

Figure 19: **This figure illustrates the security vulnerability of DEUCE-based ORAM, wherein DEUCE prevents re-encryption of all the blocks on path-6 except the evicted real blocks. This allows the adversary to identify the evicted real blocks on the path.**

NVM energy and improving NVM lifetime. Although all these works utilized the concept of PLE, their novelty was to leverage it in unique contexts and/or to improve over prior work that used the principles of PLE. However, an adoption of prior PLE-based techniques to ORAM exposes the real (i.e., data) blocks (as demonstrated with an example below), compromising the security of baseline Path ORAM. The novelty of LEO is not just in recognizing this security vulnerability, but also in crafting a secure, efficient ORAM encryption framework to address this security flaw.

Consider the following example that illustrates the security vulnerabilities of adopting PLE-based schemes (DEUCE in this example) in ORAM and how LEO addresses these vulnerabilities. Figure 18 illustrates the internal state of the Path ORAM tree and ORAM controller after the read phase of an ORAM access, prior to the path write. In this illustration, path-6 is being accessed. The stash has real blocks with logical addresses (LAs) 1, 2, and 3. The position map (PosMap) shows that the real blocks with LAs 1, 2, and 3 are mapped to leaves 6, 7, and 7, respectively. The buckets B_1, B_2, and B_3 on path-6 at have 2, 2, and 1 empty or dummy slots, respectively.

DEUCE-based ORAM: Figure 19 illustrates the security vulnerability of integrating DEUCE in Path ORAM by tracking the internal state of the Path ORAM tree and the ORAM controller after the path write. During the path-6 write, the blocks with LA 1, LA 2, and LA 3 are evicted from the stash to the empty slots in B_3, B_2, and B_2 respectively. For this illustration, let us assume that all the pre-existing real blocks on the path return to their previous bucket slots. Therefore,

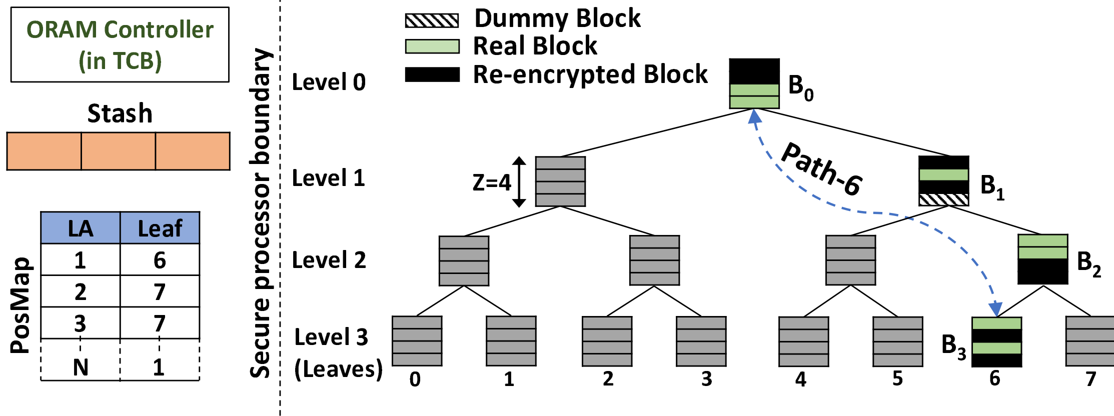

Figure 20: **LEO ensures MR_{MAX} re-encryptions in all buckets on a path write; if a bucket has less than MR_{MAX} evicted blocks, it randomly selects pre-existing real or dummy blocks for re-encryption. For path-6, MR_{MAX}=2, and LEO ensures 2 re-encryption in all buckets. Since LEO re-encrypts both real and dummy blocks, the attacker cannot identify the evicted real blocks on the path without prior knowledge of the secure internal state of the ORAM tree.**

only the evicted real blocks (solid black blocks) are the modified blocks on the path write. By design, DEUCE avoids re-encryption of the unmodified blocks; hence, only the evicted real blocks are re-encrypted. As a result, an adversary monitoring the changes on the path can identify the re-encrypted blocks as real blocks, thereby compromising the Path ORAM security guarantees.

LEO: Figure 20 illustrates how LEO overcomes the security vulnerabilities of DEUCE-based ORAM (PLE-based scheme in general) by mandating the same block re-encryption count in each bucket on a path write; this count equals the highest number of evicted blocks (i.e., MR_{MAX}) at any bucket on that path write. However, if MR_{MAX} blocks were not evicted to a bucket, random pre-existing blocks (irrespective of their real or dummy status) are selected to achieve MR_{MAX} re-encryptions in that bucket; therefore, to the adversary, the re-encrypted blocks can be real or dummy blocks with equal probability. In the example, during the path-6 write, the blocks with LA 1, LA 2, and LA 3 are evicted to B_3, B_2, and B_2 respectively. Similar to the discussion above for DEUCE-based ORAM, assume that all the pre-existing real blocks on the path return to their previous bucket slots. The MR_{MAX} of path-6 is 2, since the highest number of evicted blocks at any bucket of path-6 is 2 (i.e., in B_2). Since LEO mandates MR_{MAX} re-encryptions in all buckets (MR_{MAX} is 2 in this example), 2, 2, and 1 random blocks are re-encrypted at B_0, B_1,

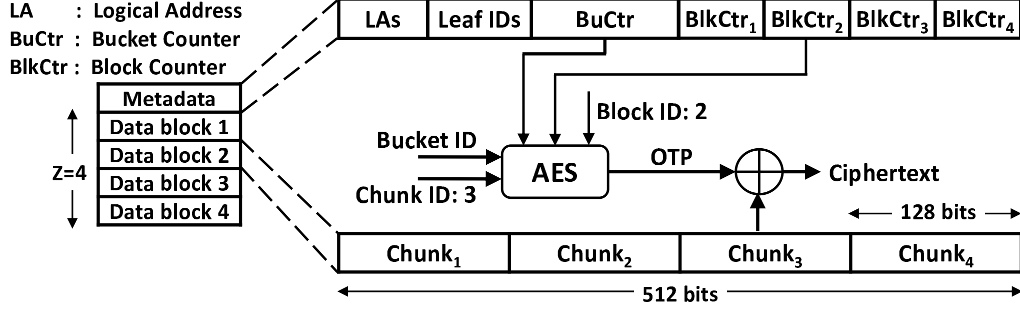

Figure 21: **Two-level counter design illustration. A bucket with Z=4 is shown, with its metadata block and block 2 expanded (not to scale). The metadata contains the LAs, the leaf IDs, a shared bucket counter (BuCtr), and 4 block counters (BlkCtrs) for 4 data blocks. To evaluate OTP for chunk 3 within block 2, BlkCtr_2 is used along with the BuCtr; block ID and chunk ID values are 2 and 3, respectively.**

and B_3, respectively. Without prior knowledge of the secure internal state of the ORAM tree, the adversary cannot identify the re-encrypted blocks as real or dummy blocks.

5.2.4 Two-level counter design

Whereas the baseline ORAM provisions only one shared 64-bit counter per bucket [87], which increases on every bucket write and alters the OTP even for the B_{SAME}/dummy blocks, LEO provisions Z counters for Z blocks in the bucket translating into Z different OTPs. However, instead of allocating Z 64-bit counters to Z 512-bit blocks for a counter memory overhead of 12.5% per block, LEO utilizes a *two-level counter design* (TCD) with lower memory overhead. In the *two-level counter design* (TCD), (i) each block in the bucket is associated with a small 8-bit block counter (BlkCtr) while (ii) a large 64-bit bucket counter (BuCtr) is shared by all the blocks, similar to the baseline ORAM, for an aggregate counter memory overhead of $\approx$ 4.7% per block. The BlkCtrs are stored in the metadata block of a bucket alongside the LAs, leaf IDs, and the BuCtr, as shown in Fig. 21. The BuCtr concatenated with BuCtr_i generates the effective counter for a block_i.

For TCD-based CME, the plaintext in the bucket is partitioned into chunks of AES block size and XORed with an OTP, similar to the baseline ORAM. LEO modifies the OTP generation to incorporate the TCD as $\text{AES}_K(\text{BuCtr}||\text{BlkCtr}_i||\text{BuID}||\text{BlkID}_i||\text{ChkID})$, where BlkCtr_i and BlkID_i ($1 \leq i \leq Z$) are the counter and unique ID of the block where the chunk resides. Using BlkID prevents OTP reuse between two blocks in the same bucket with similar BlkCtr values. Figure 21

illustrates OTP generation for chunk 3 in block 2 of a bucket using TCD. For block 2, LEO uses BlkCtr_2 along with the BuCtr; the BlkID and ChkID are 2 and 3, respectively.

The BlkCtr is incremented by 1 in the write phase if it is a $B_{\text{NEW_EV}}$ or B_{DIFF} block, or a B_{SAME}/dummy block randomly selected to ensure MR_{MAX} re-encryptions for the bucket. If any BlkCtr overflows, the bucket MR_{COUNT} is set to Z, all Z BlkCtrs are reset, and the BuCtr is incremented by 1 to avoid counter reuse. Without BlkCtr overflow, the BuCtr and BlkCtr for (Z-MR_{MAX}) B_{SAME}/dummy blocks are unchanged, translating to unmodified OTPs. The unmodified OTPs prevent re-encryptions for (Z-MR_{MAX}) B_{SAME}/dummy blocks in each bucket, eliminating associated NVM writes. Following state-of-the-art [87], the bucket and block counters are stored in plaintext in NVM.

LEO's novelty is not in proposing TCD, but using the general split counter design (SCD) [82] approach to enable the novel secure selective-encryption of blocks in a bucket on a path write. The primary differences between TCD and SCD proposed in [82] are as follows:

- Although TCD appears to be structurally similar to SCD, they serve entirely different objectives. Whereas SCD was proposed to reduce counter overflow overheads and improve the efficiency of counter caching in counter mode encryption (CME) of data in memory, TCD is used in LEO to reduce redundant re-encryption of unmodified blocks.

- Naive adoption of the SCD in baseline Path ORAM results in security vulnerabilities equivalent to the adoption of PLE schemes, as explained in Section 5.2.3. The integration of SCD culminates in selective re-encryption of only real evicted blocks on a path write, enabling the adversary to distinguish between real and dummy blocks, and also the accessed/evicted buckets. In contrast, apart from the real evicted blocks, TCD enables LEO to randomly re-encrypt both pre-existing real and dummy blocks on a path write. The attacker cannot identify the real blocks on a path without prior knowledge of the secure internal state of the ORAM tree.

LEO ensures secure reduction of redundant block re-encryptions; however, as demonstrated, the CME framework of the baseline Path ORAM cannot support LEO. Therefore, the TCD architecture is explicitly developed to integrate LEO in Path ORAM.

5.2.5 Security: Overview

The baseline ORAM prevents information leakage about the (i) LA accessed, memory access type, and associated data, (ii) bucket holding accessed LA block or $B_{\text{NEW_EV}}$ blocks, and (iii) distinction between data and dummy blocks; the baseline ORAM ensures negligible stash overflow probability for security [34].

LEO preserves the security guarantees of the baseline ORAM. First, LEO adopts measures identical to the baseline ORAM to obfuscate the plaintext data, the memory access type, and the accessed LA. LEO keeps data encrypted, performs a read and write on every ORAM access, and utilizes a secure random leaf label remapping for LAs, identical to the baseline ORAM. Therefore, LEO prevents information leakage about the data, type, or address of memory access. Second, since LEO ensures MR_{MAX} re-encryptions for all buckets on an accessed path, it prevents identification of the bucket holding the desired LA or the $B_{\text{NEW_EV}}$ blocks. Third, LEO does not allow the attacker to distinguish between data and dummy blocks. The re-encrypted blocks may be $B_{\text{NEW_EV}}/B_{\text{DIFF}}$ data blocks, or randomly selected B_{SAME} data blocks or dummy blocks with equal probability; the blocks not re-encrypted can be B_{SAME} data blocks or dummy blocks with equal probability, thwarting the efforts to differentiate between data and dummy blocks. Finally, stash overflow depends on the stash size, eviction algorithm, and the number of ORAM levels [34]. Since LEO utilizes the same public stash eviction algorithm, identical stash size, and same number of ORAM levels as the baseline ORAM, it is guaranteed to not affect the probability of stash overflow. Furthermore, although LEO reveals path MR_{MAX}, it does not compromise baseline ORAM security. MR_{MAX} is the maximum MR_{COUNT} of buckets on a path; MR_{COUNT} of each bucket is a feature of the public stash eviction algorithm and the secure random mapping of data blocks to leaves. Since MR_{MAX} is derived from secure/public mechanisms of the baseline ORAM, it does not leak additional information.

5.2.6 Security: Detailed discussion

Real block deduction: The baseline Path ORAM eviction scheme proactively pushes real (i.e., data) blocks towards the leaves of the ORAM tree, and these evicted blocks are re-encrypted before being written back to the NVM. Therefore, it might appear that by selective encryption, LEO

can reveal the evicted real blocks. Although LEO reduces the number of re-encryptions on a path write (i.e., write phase of an ORAM access), it re-encrypts both real and dummy blocks, ensuring that the real blocks cannot be deduced in any manner as follows. LEO ensures that all the buckets on a path enforce MR_{MAX} (and not MR_{COUNT}) block re-encryptions, with mandatory re-encryptions of the evicted real blocks in a bucket. Note that if the MR_{COUNT} of a bucket is less than MR_{MAX}, LEO re-encrypts (MR_{MAX}-MR_{COUNT}) random real or dummy blocks in that bucket, for a total of MR_{MAX} re-encryptions for that bucket (please refer Sec. 5.2).

For an adversary without prior knowledge of the secure internal state of the ORAM tree, a re-encrypted (updated) block in a bucket can be a real block *or* a randomly selected dummy block to attain MR_{MAX} re-encryptions for that bucket; therefore, the probability of the updated block being real *or* dummy is equal. Similarly, the non-updated blocks can also be dummy blocks *or* pre-existing real blocks that were unmodified and returned to the same buckets from which they were fetched in the read phase. This is equivalent to the baseline Path ORAM wherein the encrypted blocks can be real or dummy with equal probability. Therefore, LEO preserves the security guarantees of the baseline, thwarting identification of real blocks.

MR_{MAX} and real block density on a path: This paragraph explicitly demonstrates that revealing MR_{MAX} of a path is secure and does not leak information about the number of real blocks on that path. In this context, it is important to note that LEO does not reveal the total number of blocks evicted from the stash on a path write. LEO only reveals MR_{MAX}, which is the maximum MR_{COUNT} over all buckets on that path during a path write. An overloaded path with only a few empty slots can experience a high MR_{MAX}; if one of the buckets (e.g., at the top of the ORAM tree) had most of the empty slots and received most of the evicted blocks, that bucket would have a high MR_{COUNT}, mandating a high MR_{MAX} for the overloaded path write. In contrast, a path with high number of empty slots can experience a small MR_{MAX}; if the stash had only few blocks to be evicted and if the leaf map of those blocks were such that each bucket on the path received at most one evicted block, then the MR_{MAX} of the path is equal to 1. Hence, a path with many empty slots can have a low number of updates. The adversary cannot infer if the path was overloaded without prior knowledge of the secure internal state of the ORAM. Hence, MR_{MAX} of a path is independent of the number of real blocks on that path and does not leak information about the real block load of that path.

MR_{MAX} and current stash state: By design, there is no direct dependence between MR_{MAX} and the stash load for the following two reasons. First, both the baseline Path ORAM and LEO adopt a proactive eviction algorithm that evicts as many blocks from the stash as possible, irrespective of the stash load. Second, LEO does not reveal the total number of blocks evicted from the stash, decoupling the number of bucket updates, i.e. MR_{MAX} of a path write, from the stash load during that ORAM access. Therefore, the number of block updates/re-encryptions on a path write does not leak information about the stash load.

In the event of high stash load resulting in high stash overflow probability on consecutive ORAM accesses, both baseline Path ORAM and LEO activate background eviction [87]. Background eviction, proposed by Ren *et al.* in [87], prevents stash overflow by utilizing dummy ORAM accesses. In a dummy ORAM access, a random path is accessed, but no real block is remapped. Hence, all the real blocks fetched in the read phase return to their respective buckets at the end of the write phase; additionally, if possible, blocks from the stash are evicted, reducing the stash load. The ORAM continues issuing dummy accesses till the stash load is reduced below a threshold. Note that these dummy ORAM accesses are indistinguishable from real ORAM accesses to ensure security [87].

5.2.7 Hardware overhead

LEO requires (i) a buffer to hold the LAs of blocks in each bucket fetched during the read phase, (ii) memory overhead to store the additional BlkCtrs, and (iii) control logic to determine the nature of the blocks (i.e., $B_{\text{SAME}}/B_{\text{NEW_EV}}/B_{\text{DIFF}}$), MR_{COUNT} of each bucket, and path MR_{MAX}. First, during ORAM write phase, the buffer is checked to ensure whether the evicted blocks are B_{SAME}, $B_{\text{NEW_EV}}$, or B_{DIFF}. If an LA (except the accessed LA on a logical write) is returned to the same bucket, it is a B_{SAME} block; else, it is a $B_{\text{NEW_EV}}/B_{\text{DIFF}}$ block. With L+1 buckets on a path, each holding Z 8-byte LAs, LEO requires $(L+1)\times Z\times 8$ bytes of buffer. In practice, for a 16GB ORAM with 26 levels and Z value of 4, the buffer size is 832 bytes. Second, LEO requires memory to support TCD. For a block/cache line size of 512 bits, an 8-bit counter results in $\approx 1.6\%$ additional memory per block over the baseline ORAM. LEO utilizes the same random number generator used for random leaf remapping to randomly select B_{SAME}/dummy blocks for re-encryptions, utilizing some logic to constrain the limit from 1 to Z. Finally, the control logic compares the LAs in

Table 4: **Configuration for evaluation setup**

MARSS+DRAMSim2		NVMain and in-house lifetime simulator	
Core	4 out-of-order, 2GHz	Memory size	16GB, 1.33 GHz
Private L1 (I/D) cache	32kB, 2-way, 2ns	Channel; rank; bank	2 channels; 2 ranks/ch; 8 x8 banks
Private L2 cache	128kB, 8-way, 5ns	Scheduling	First ready-first come-first serve
Shared L3 cache	4MB, 32-way, 20ns	Row buffer management	Open page policy
DDR parameters modified following [1], 1.33GHz		Lifetime; wear-levelling (WL)	10^8 writes per cell [1]; uniform WL

the buffer from (i) to the LAs being written to buckets, determining the $B_{\text{SAME}}/B_{\text{NEW_EV}}/B_{\text{DIFF}}$ blocks, the MR_{COUNT} of each bucket, and the path MR_{MAX}. A 5-stage tournament setup is used to calculate the maximum MR_{COUNT}, i.e., MR_{MAX}, from 26 MR_{COUNT} values. The control logic is designed and synthesized at an estimated logic overhead of $\approx$ 4k 2-input nand gates.

5.3 EVALUATION AND RESULTS

LEO is compared to the baseline Path ORAM [34] and evaluated on an SLC PCM architecture with the SPEC CPU2006 benchmark suite [23]. The evaluations utilize (i) trace-based simulations for NVM energy and lifetime evaluations with NVMain [26] and an in-house simulator, respectively, and (ii) full-system simulations to evaluate overall system IPC. Multi-billion instruction memory traces were generated using Intel Pin [27] binary instrumentation tool, and used for evaluating NVM energy and lifetime. For full-system simulations,MARSS [28] coupled with DRAMSim2 [88] (parameters configured to represent an NVM) is utilized for monolithic, cycle-accurate system simulation. For evaluations, identical benchmarks are executed on each core of the simulated system. The configuration parameters are summarized in Table 4. Both trace-based and full-system simulations use SLC PCM energy and latency values from [91].

The parameters of the evaluated ORAM architectures are representative of the optimal configurations obtained through design exploration in [87] and used widely in [35, 37–39, 87]. An 8GB ORAM with Z=4 and stash size of 200 is used for evaluations. Both the baseline and LEO assume 50% utilization, i.e., upto 50% of the NVM is assumed to hold data blocks. Therefore,

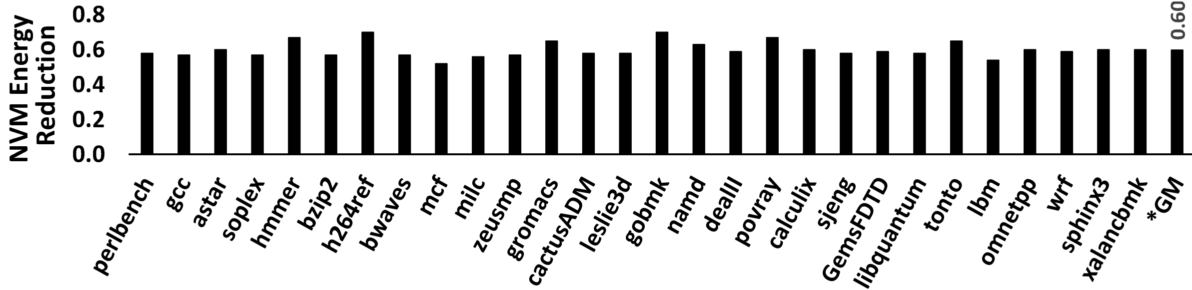

Figure 22: **NVM energy reduction of LEO (normalized to the baseline) evaluated on SPEC CPU2006 benchmarks [23] using NVMain [26]. For SLC PCM, LEO reduces NVM energy, on average, by 60% over the baseline.**

an 8GB ORAM requires 16GB NVM resulting in a ORAM tree of 26 levels. Both the baseline and LEO integrate Fork Path [38], and also implement tree-top caching [39] of 1MB, effectively storing 12 levels from the root on-chip in the TCB. The address mapping scheme of "row:bank:column:rank:channel" with a sub-tree layout approach is also adopted from [87].

5.3.1 NVM energy and lifetime

As shown in Fig. 22, LEO reduces NVM energy on average by 60% over the baseline ORAM. With Z=4, the results show that LEO reduces the average block re-encryptions per bucket from 4 to 1.5; this includes the block counter rollover requiring re-encryption of the entire path. With lower block re-encryptions, LEO achieves a lower cell-write rate because (Z-$\mathrm{MR_{MAX}}$) B_{SAME} and/or dummy blocks are not re-encrypted in each bucket, reducing the number of cache lines written with re-encrypted data, decreasing NVM write energy in practice. In Fig. 22, every workload experiences almost similar reduction in block re-encryption, with marginal variations due to their varying working set sizes (WSS). Lower WSS of benchmarks like gobmk and povray leaves more dummy blocks in the ORAM, translating to higher reduction in block re-encryption.

Similarly, Fig. 23 shows that LEO improves the average memory lifetime by 1.51× over the baseline. Since lower block re-encryptions lead to reduced NVM writes, fewer cells need to be programmed during the write phase, reducing the wear rate of memory. Similar to NVM energy, workloads with lower WSS experience marginally better reduction in NVM writes, yielding better improvement in lifetime.

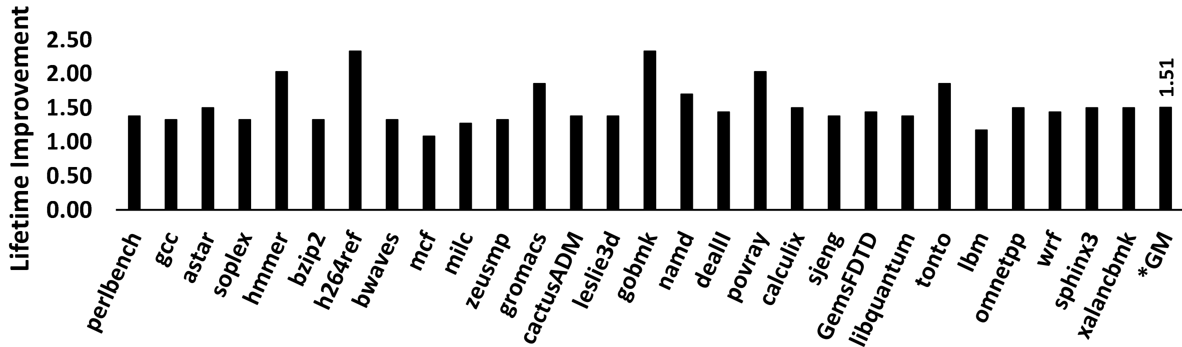

Figure 23: **NVM lifetime improvement of LEO (normalized to the baseline) evaluated on SPEC CPU2006 benchmarks [23] using NVMain [26]. For SLC PCM, LEO increases NVM lifetime, on average, by 1.51× over the baseline.**

5.3.2 System IPC

As shown in Fig. 24, LEO improves average system performance (reported as speedup) by 9% over the baseline. LEO reduces the write traffic to NVMs, enabling faster write phase completion on ORAM accesses, reducing overall ORAM access latency. Since the write phase is accelerated, it leads to lower service timing of the succeeding ORAM accesses. In practice, the blocks of a bucket are striped across the memory ranks due to the address mapping, enabling parallel writes of re-encrypted blocks. Since LEO reduces writes to unmodified blocks, the free parallel slots over memory channels are utilized to write modified blocks of the next bucket in the absence of bank conflicts. Since the write phase is not on the critical path of program execution, accelerating it does not improve system IPC significantly. LEO shows better speedup in system performance of benchmarks with high memory access rate (measured as misses-per-kilo-instruction (MPKI)) like mcf and milc is more dependent on ORAM access latency than low MPKI benchmarks.

5.4 RELATED WORK

Among architecture-based solutions to improve ORAM efficiency [35, 37–39, 87], memory write reduction approaches are most relevant to this work. Fork Path [38] utilizes path merging to prevent write-back of overlapping buckets between successive ORAM accesses; the evaluations inte-

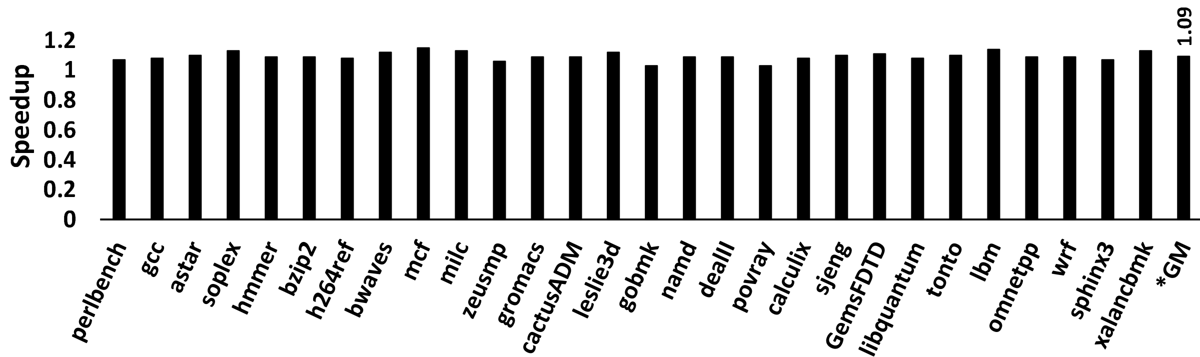

Figure 24: **Speedup (i.e., increase in IPC) of LEO (normalized to the baseline) evaluated on SPEC CPU2006 benchmarks [23] using MARSSx86 [28] coupled with DRAMSim2 [88]. For SLC PCM, LEO improves performance, on average, by 9% over the baseline.**

grate Fork Path in both the baseline and LEO. Flat ORAM reduces a path write to a single block write [77]; however, the threat model for Flat ORAM is weaker than LEO, considering adversaries capable of monitoring only write accesses. Recent work on access obfuscation in 2.5D/3D memories [2, 92] extend the TCB to include the logic on memory module, leaving only the memory bus susceptible to attacks. This allows low overhead access obfuscation without ORAM integration by simple encryption of command/address/data on the bus; however, these are applicable only to 2.5D/3D memories with trusted encryption/decryption and attestation logic on the DIMMs. LEO considers a more conservative threat model and can be seamlessly integrated with conventional DIMMs without trusted encryption/decryption logic overhead on the DIMMs.

5.5 CONCLUSIONS

LEO is a secure, optimal encryption architecture for cost-effective integration of ORAM with NVMs to thwart access-pattern-based data confidentiality attacks. LEO reduces redundant re-encryptions of unchanged blocks during the write phase of an ORAM access, which reduces expensive NVM writes in practice. LEO ensures security equivalent to the baseline ORAM by mandating similar block re-encryption count in all buckets on an accessed path, equal to the highest number of modified blocks in an individual bucket during that ORAM access. LEO uses a two-level counter design to realize this efficient re-encryption framework that decreases NVM energy, improves lifetime, and enhances overall system performance.

6.0 ORAM: DISCUSSION AND LOOKING FORWARD

This chapter focuses on directed discussions regarding the domain of memory authentication and access-pattern obfuscation, highlighting the impact and relevance of the solutions presented in this dissertation. The first section discusses the practical aspects and commercial potential of ORAM. The second section discusses the implications of eliminating a key assumption of a secure OS from the threat model on classic authentication and obfuscation solutions. The third section discusses the potential modifications to traditional security solutions for adapting to hybrid main memories. Finally, the fourth section discusses the necessity of authentication and obfuscation solutions in the broader computing community.

6.1 PRACTICALITY AND COMMERCIALIZATION OF ORAM

Oblivious RAM (ORAM) primitives are effective solutions for access-pattern-based attacks on data confidentiality. However, due to the amplification of a single memory request into multiple memory reads **and** writes, ORAM incurs significant performance overhead of approximately $4\times$. This dissertation proposes ReadPRO that improves the system IPC by 38% over baseline Path ORAM by efficient read prioritization; however, ReadPRO still incurs $\approx 2.5\times$ system performance overhead in comparison to a system that does not integrate ORAM primitive for eliminating the access-pattern side channel. This considerable performance overhead to ensure data confidentiality against side-channel attacks might pose questions about the viability of adoption in practical memory systems and their eventual commercialization.

Whereas ORAM decreases system performance, it is currently the state-of-the-art solution (specifically Path ORAM) for ensuring security against access-pattern-based side channel attacks in

the secure processor paradigm. The wide adoption of cloud computing and storage has drastically increased the attack surface in the computing landscape, exposing multiple points of attacks in the datacenters that cannot be secured by a client. Additionally, recent attacks reveal that memory access patterns can be conveniently leveraged; for e.g., [93] identify the associated encrypted data/encryption key through monitoring just write access patterns. Therefore, any application or user that requires protection against access-pattern-based attacks should mandatorily integrate ORAM.

Although there is a burgeoning interest in architecting efficient ORAM solutions for main memory systems following the introduction of Path ORAM [34], the research is still in its nascent stage and is expected to provide rapid improvements to the performance challenges of ORAM. The stage is similar to the initial state of research in memory encryption and memory authentication, wherein the system performance overhead in terms of execution time increased upto 150% for SPEC CPU2006 benchmarks [11]. However, successive work on efficient memory authentication, specifically Bonsai Merkle Tree (BMT) decreased the performance overhead significantly to 13%. A similar trend of improvement is expected for ORAM considering the close resemblance of the objective and similarity in the basic building-blocks, coupled with the active interest of the architecture research community in this field.

Whereas current ORAM solutions consider the entire memory to be protected by ORAM, future designs might consider a modular design to reduce ORAM overheads; only the memory regions which require access pattern obfuscation should be protected by ORAM, similar to the idea of protected memory region (PRM), i.e., enclaves in commercially available Intel SGX solution for memory integrity and authentication [94]. Orthogonally, address obfuscation solutions for 3D stacked memories like Micron HMC or Samsung HBM propose and utilize an extended TCB for reducing the security overheads [2, 92]. The solutions in [2, 92] primarily utilize the integrated computation logic in 3D stacked memories to implement trusted logic on the DIMM, and perform address/command encryption/decryption to prevent exposure of address/commands in plaintext, effectively eliminating the plaintext access pattern side channels. However, as discussed in [3, 4] due to low yield, limited capacity per package, and high cost, these active memory solutions are more suitable as an off-chip cache to support a passive main memory, which will require ORAM to realize access pattern obfuscation. The ORAM solutions for passive memory in [3, 4]

leverage secure chip-on-DIMM architectures to offload the ORAM controller to the DIMM, reducing the bandwidth burden on the processor-memory channel and increasing parallelism in ORAM accesses. Whereas these novel architectures effectively reduce ORAM overheads, the system designers now need to trust both the CPU and DIMM vendors. Although the CPU vendors can design custom DIMMs with a secure-chip to reduce the number of trusted parties, this approach reduces the flexibility to choose CPU and DIMM from different vendors and affects market competition.

The true viability of a security solution is established by its widespread adoption in industry. Memory encryption and authentication solutions have been accepted in industry as an essential memory design feature, as evident in Intel SGX and ARM TrustZone [95]. However, the industry adoption of these elementary cryptographic primitives lagged behind the academic research timeline of this domain; whereas BMT was proposed in 2007 [11], Intel SGX was introduced approximately around 2013. Following a similar trend, the industry adoption of ORAM can be expected to lag behind the academic research; however, due to the present computing landscape with an increased attack surface, ORAM or some variant for access pattern obfuscation will be essential in industry to ensure confidentiality. Recent work have architected ORAM primitives that integrates seamlessly with commercially available Intel SGX solutions [96], paving a way forward for industry adoption.

6.2 TIGHTER TCB WITH UNSECURE OPERATING SYSTEM (OS)

The majority of memory security solutions for encryption and authentication consider a primary assumption of the operating system (OS) being secure. Specifically, the critical OS components like building and managing page table are assumed to be secure. A corrupt OS can easily change the page table information; this results in wrong data being fetched for a virtual address generated by the application, effectively resulting in a splicing attack. OSes are large complicated software systems with numerous demonstrable security vulnerabilities that have been exploited over decades. Therefore, it is important to develop memory security solutions that does not assume a secure OS.

Currently, there are academic and commercial solutions that provide hardware-driven memory integrity guarantees in the presence of compromised OS. In academia, the memory security solution in AEGIS [97] builds a Merkle Tree over the virtual address space (VAS), defined as VAS

tree. Although VAS tree protects against corrupt OS, it imposes a large overhead since the VAS is much bigger (64-bit space) than the actual physical address space (PAS). In reduced address space (RAS) tree solution [83], each page allocated in memory is provided a unique number that represents the RAS, and the mapping is stored securely on-chip, which is inaccessible by the OS. An MT is dynamically built on top of the RAS, which results is significantly smaller overheads. In Intel SGX, a commercially available and widely used memory security solution, a protected memory region (PRM) is reserved, which is managed by trusted hardware, without any OS involvement. Since the PRM is small, the integrity tree built on top is small; currently, the limit of the PRM is 128MB. Therefore, all the solutions which maintain memory integrity in presence of corrupt OS derive their security guarantees bootstrapped from a trusted hardware, which is not managed by the OS. This dissertation presents a low overhead memory authentication solution, ASSURE, which is easily integrable with VAS tree or Intel SGX, since ASSURE intrinsically does not rely on the OS or any address translation mechanism.

Although state-of-the-art ORAM primitives assume a secure OS, future secure memories must realize secure ORAM implementations without secure OS provisioning. The PosMap in the Path ORAM maps the logical address to leaf labels; hence, the security guarantees depend on the OS-controlled translation from virtual to logical address. Since the problem domain is similar to memory authentication, techniques similar to RAS is a potential solution towards a secure ORAM with unsecure OS. Similar to RAS, all the allocated active pages in memory can be allotted a unique page number, maintained by a trusted hardware on the processor chip; the PosMap should utilize this unique number, eliminating the dependency of the ORAM algorithm on OS.

6.3 AUTHENTICATION/OBLIVIOUS ACCESS IN EMERGING HYBRID MEMORIES

To meet the performance and reliability requirements of modern memory systems, hybrid DRAM-NVM memory architectures have emerged as a more practical approach towards NVM adoption in main memory [98–107]. The DRAM system filters a majority of the write operations, effectively concealing the high write latency/energy and increasing the lifetime of the NVM system. Hybrid DRAM-NVM architectures are broadly categorized into two approaches: (i) DRAM as cache with NVM (DC-NVM), and (ii) DRAM in parallel with NVM (DP-NVM). In DC-NVM,

DRAM is utilized as an OS-transparent cache in front of NVM main memory as an interface to the processor [98, 100–103]. In DP-NVM, DRAM and NVM are at the same level in the memory hierarchy, with the operating system (OS) managing them under a single physical address space [99, 104–106].

In both DC-NVM and DP-NVM, the DRAM and NVM are unsecure. Whereas the DRAM is treated as a cache in DC-NVM, it should be mandatorily authenticated using a Merkle tree; the tags and meta-data should also be authenticated. Although the DRAM is protected by an MT, the NVM data can be tampered; hence, the NVM should be integrity-protected utilizing a separate MT. The independent roots of the DRAM and NVM MT should be securely stored on the processor. Similar to the traditional DRAM-/NVM-only main memories, the performance if memory authentication can be improved with on-chip caching of the MT nodes and efficient MT solutions like ASSURE. Similar to DC-NVM, DP-NVM should provide integrity protection schemes to both NVM and DRAM. Since the DRAM and NVM belong to the same address space in DP-NVM, they should be protected by a single MT, and efficient MT solutions like ASSURE can be implemented for reducing MT overheads. The insights for authentication in DC-NVM and DP-NVM can be translated to access obfuscation since Path ORAM has a similar tree structure as an MT. Whereas in DC-NVM, two independent ORAMs should be assigned to the DRAM and NVM, a single integrated ORAM can secure DP-NVM against access-pattern side channels. Note that both ReadPRO and LEO can be integrated with DC- and DP-NVM Path ORAMs for low overhead access-obfuscation.

3D integration has facilitated the fabrication of smart memories equipped with logic on the DIMM, facilitating considerable processing-in-memory capabilities [2, 92, 108–111]. A typical smart memory like hybrid memory cube (HMC) is composed of several vertically integrated stack of DRAM dies with a logic layer at the bottom and all the layers connected using through-silicon-vias (TSVs). Current NVM prototypes are also integrating considerable logic within the DIMM to support wear leveling, scheduling, and failure re-mapping, etc [92]. Recent work on memory security consider the logic on the smart memory DIMM to be trusted and include it in the TCB; only the processor-memory bus is considered to be unsecure. Although the logic in current smart memory DIMMs only consists of essential circuitry, it can include cryptographic hardware considering its area and thermal power budget of around 55W [112]. This cryptographic hardware on the trusted

logic facilitates the deployment of authenticated encryption using Galois Counter Mode, eliminating the requirement of MT and significantly reducing authentication overhead [2]. In GCM, the processor and memory shares an encryption key and encryption counter. The sender generates an authentication tag based on the encrypted data, the shared counter, and a secret encryption key. The receiver regenerates the tag from the received encrypted data, the shared counter, and secret key, accompanied by a validation against the received tag; the equality of the tags ensure integrity of the data on the channel. A recent work, STASH [113] presents the first security work on smart hybrid memories, wherein low overhead solutions are proposed to achieve encryption, authentication, and access-pattern obfuscation. STASH utilizes a page-level MT, recovery-compatible MT updates, and page migration friendly counter management to reduce memory overhead by 12.7×, increase system performance by 65%, and improve NVM lifetime by 2.5×.

6.4 AUTHENTICATION AND OBFUSCATION: A BROADER PERSPECTIVE

This dissertation has primarily focused on discussing the technical underpinnings of data authentication and oblivious access in computer memories. This section briefly explores the significance of the security solutions analyzed in this dissertation to other relevant domains. Since this dissertation mainly addresses data security, all data-centric enterprises are potential beneficiaries of solutions described in this dissertation. For example, financial and medical enterprises are data driven, and rely on accuracy and confidentiality of data for smooth and legal operations. Most medical and financial institutions store massive amounts of data (patient record, transaction details) in their private data centers, or more commonly, public cloud storage services [114, 115]. Public third-party cloud storage services like Amazon Web Services (AWS) or Microsoft Azure offer high availability and low latency large scale data storage, without the hassle and high cost of in-house data storage and management. However, the primary concern of outsourcing data storage is enforcing necessary data security. The data needs to be encrypted to prevent leakage of plaintext data to unauthorized parties. Data integrity is also of utmost importance; for example, the amount stored in a customer's bank account must remain unmodified over time and should be modified only on authorized accesses. The data access patterns between the client and the data center must also be obfuscated to prevent access-pattern-based attacks. A successful access-pattern-based at-

tacks can completely undo the security provisions of data encryption, leaking confidential data (for e.g., customer medical records) to unauthorized third parties, exposing both the client enterprise and the data hosting service to crippling privacy lawsuits. Therefore, access obfuscation solutions like ORAM are extremely crucial for the proper functioning of the cloud service providers and consumers. In summary, in this age of big data, most organizations rely of large scale data storage and analysis for their operations, hence, data security solutions covered in this dissertation is of utmost importance to these enterprises.

7.0 FUTURE WORK

Future research directions will focus on (i) exploring efficient solutions for ORAM read phase optimization and secure ORAM resizing, (ii) investigating the security challenges of emerging processing-in-memory architectures that require plaintext data on the memory modules for processing purposes, and (iii) investigating the interplay of security primitives with reliability enhancing architectures, focusing on leveraging the reliability improvement techniques to efficiently reduce security-related overheads.

7.1 IMPROVING ORAM EFFICIENCY

ORAM integration for access-pattern-based obfuscation in main memories incurs considerable overhead in system performance, memory energy, and memory lifetime (for NVM-based memory systems). In comparison to a non-secure baseline system, even efficient ORAM constructs proposed in [35–38, 41] incur 10-100× increase in memory traffic, 5-40× increase in access latency, and 2-10× degradation in system performance. Although the solutions proposed in this dissertation, ReadPRO and LEO, significantly decrease these overheads, the gap between performance of a memory system integrating ORAM and unsecure memory can be bridged further.

ReadPRO scheduling primarily re-orders the sequence of path accesses dependent on the type (i.e., read/write) of the original memory access. Following up this solution, the effects of re-ordering the node accesses within a single path access requires exploration, while preserving the ORAM security guarantees. The read phase of an ORAM access fetches all blocks from every bucket on the path corresponding to the mapped leaf, traditionally from the root (level 0) to the leaf (level L) [4, 34, 116]. The *root-to-leaf* scheduling is very effective in the presence of tree-top ORAM caching, wherein the top ORAM tree levels are stored on the processor chip in an ORAM cache [31, 38, 39]. Root-to-leaf scheduling first checks the buckets in the ORAM for the target data

block. If the target block is present in the cached buckets, it can be forwarded faster to the processor, because the on-chip cache read latency is lower than the memory read latency. However, the majority of target data blocks are located outside the ORAM cache, closer to the leaves (in the main memory), primarily due to (i) the small ORAM cache size in comparison to the total ORAM size and (ii) the leaf affinity of the stash eviction algorithm, wherein the data blocks in the stash are evicted as close to the leaves as possible. Some promising solutions include (i) effective secure caching mechanisms for the ORAM data that supports faster critical data fetch during a path access, and (ii) secure scheduling of node accesses from the ORAM to decrease data fetch latency when the critical data block in in the external memory.

Orthogonally, dynamic resizing of ORAM will be explored to reduce effective memory traffic over the processor-memory channels. One of the crucial features of state-of-the-art ORAM solutions for main memories is that the ORAM size is constant throughout program execution [31, 34–40]. However, the memory footprint, i.e., the resident set size (RSS) varies across (i) different workloads and (ii) different execution phases within the same workload (for example, in SPEC CPU2006 benchmarks [117]). Whereas the RSS of the workloads fluctuates, memory traffic, i.e., the number of blocks read/written back in an ORAM access is constant in state-of-the-art ORAM solutions. The memory traffic depends on the number of tree levels, and due to the constant size of ORAM prevalent in state-of-the art ORAMs, the number of tree levels are also constant, resulting in constant memory traffic. Therefore, workloads with memory footprints lower than the total address space (throughout or in certain phases of program execution) incur higher-than-necessary ORAM access overheads. Although dynamic tree sizing solutions proposed in ASSURE solves a similar problem for MT memory authentication, it assumes memory access locality in the external memory; an ORAM is specifically designed to obfuscate memory access pattern, and thereby memory access locality to the external unsecure memory. This motivates the development of a secure ORAM tree resizing algorithm and architecture to dynamically scale memory traffic. Previous work proposed a Resizable ORAM [118] for tree-based ORAM constructions; however, the resizable ORAM construct is incompatible with state-of-the-art compressed PosMap architecture proposed in [37]. The architectural and security challenges of integrating resizable ORAM with compressed PosMap will be investigated to propose efficient solutions to to enable secure dynamic resizing in state-of-the-art ORAM.

7.2 SECURITY FOR PROCESSING-IN-MEMORY ARCHITECTURES

Modern applications in domains like machine learning, graph processing, and other similar fields operate on very large datasets; additionally, these applications demonstrate low inherent spatial and temporal data locality, rendering caching ineffective [55, 56]. The applications need to read/write large amounts of data from/to memory and are bound by the processor-memory channel bandwidth. However, recent memory technology advancements have enabled 3D-stacked memories, where layers of memory dies are stacked, connected by through-silicon-vias(TSVs), with a logic layer at the bottom (for e.g., hybrid memory cube (HMC) and high bandwidth memory (HBM)). Processing-in-memory (PIM) architectures offload some of the tasks that generate heavy memory traffic to the logic layer in 3D memory, thereby avoiding the data transfer delay across the memory channel [44–54, 57].

Although PIM enables processing near-data processing, it requires the data to be in plaintext on the unsecure external memory. Although previous work like InvisiMem [2] and ObfusMem [92] provided security solutions for 3D stacked memories, they did not consider PIM architectures and keep data encrypted in memory. Therefore, to enable PIM, there are 2 options: (i) Implement homomorphic encryption (HME), where the logic layer can perform simple operations with encrypted data (i.e., without decryption); however, HME incurs orders of magnitude higher latency and dynamic energy, negating all the advantages of PIM. (ii) Perform data decryption on the external memory and utilize it for computations without leaking information. This motivates a detailed exploration of plaintext data security in PIM architectures, and development of efficient security solutions to prevent reduction in the benefits of PIM.

7.3 SECURITY-RELIABILITY CO-DESIGN

Recent work has demonstrated that co-design of security and reliability constructs improves the system performance by utilizing authentication constructs to perform efficient error correction in DRAM-based systems [59]. In [59], the ECC chip on a DIMM is repurposed to store MACs, which can now be accessed in parallel to the data, reducing additional memory accesses to fetch metadata for memory authentication. However, [59] is developed around ECC-based reliability systems in

DRAM memories. As demonstrated in multiple previous work, the error characteristics in NVM-based systems is significantly different than DRAM-based systems: (i) hard failures are dominant in NVMs, so dedicated ECC bits are required for them, (ii) the frequency of errors in a single cache line is significantly higher than what can be handled by standard SECDED codes, (iii) the hard errors are localized in certain memory regions within a page and within a cache line, enabling better allocation of resources; these factors motivated the development of a vast array of NVM reliability improvement techniques over the past few years [21, 60–69]. The effects of reliability improvement solutions for NVM systems on the security of NVM memory systems should be evaluated, thereby developing insights for architecting efficient reliable and secure NVM systems.

Additionally, the integration of hard-fault tolerance techniques with NVM ORAMs will be explored. Due to the significant increase in memory traffic, each ORAM access writes more blocks (i.e., cache lines) than a normal memory access in unsecure memory. This increased write frequency will deteriorate the memory lifetime of NVM ORAM implementations, necessitating provisioning of proportional error correction resources (for e.g., error correction pointers, or ECPs). Previous work demonstrates that equal distribution of error correcting resources across all blocks is inefficient [119, 120], motivating development of smart provisioning of these resources across memory domains that experience high write frequency. However, in the ORAM tree, all paths are accessed with equal probability, without the accesses being focused on any subtree. In future, novel ways for efficient and secure allocation of error correction resources can be explored at various ORAM levels to enable ORAM integration with NVM memory systems feasible.

BIBLIOGRAPHY

[1] J. Lee, T. Kim, and J. Huh, "Reducing the memory bandwidth overheads of hardware security support for multi-core processors," *IEEE Trans. on Computers*, vol. 65, no. 11, 2016.

[2] S. Aga and S. Narayanasamy, "InvisiMem: Smart memory defenses for memory bus side channel," in *Proc. International Symposium on Computer Architecture*, 2017.

[3] R. Wang, Y. Zhang, and J. Yang, "D-ORAM: Path-ORAM delegation for low execution interference on cloud servers with untrusted memory," in *Proc. International Symposium on High Performance Computer Architecture*, 2018.

[4] A. Shafiee, R. Balasubramonian, F. Li, and M. Tiwari, "Secure DIMM: Moving ORAM primitives closer to memory," in *Proc. International Symposium on High Performance Computer Architecture*, 2018.

[5] R. B. Lee, "Security basics for computer architects," *Synthesis Lectures on Computer Architecture*, vol. 8, no. 4, 2013.

[6] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling," in *Proc. International Symposium on Microarchitecture*, 2009.

[7] M. K. Qureshi, A. Seznec, L. A. Lastras, and M. M. Franceschini, "Practical and secure PCM systems by online detection of malicious write streams," in *Proc. International Symposium on High Performance Computer Architecture*, 2011.

[8] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *Proc. International Symposium on Computer Architecture*, 2014.

[9] M. Zhang, L. Zhang, L. Jiang, Z. Liu, and F. T. Chong, "Balancing performance and lifetime of MLC PCM by using a region retention monitor," in *Proc. International Symposium on High Performance Computer Architecture*, 2017.

[10] G. E. Suh, D. Clarke, B. Gassend, M. v. Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *Proc. Intl. Symposium on Microarchitecture*, 2003.

[11] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "Using address independent seed encryption and Bonsai Merkle Trees to make secure processors OS- and performance-friendly," in *Proc. Intl. Symposium on Microarchitecture*, 2007.

[12] A. D. Hilton, B. Lee, and T. Lehman, "PoisonIvy: Safe speculation for secure memory," in *Proc. Intl. Symposium on Microarchitecture*, 2016.

[13] W. Shi, H.-H. S. Lee, M. Ghosh, C. Lu, and A. Boldyreva, "High efficiency counter mode security architecture via prediction and precomputation," in *Proc. International Symposium on Computer Architecture*, 2005.

[14] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," in *Proc. Intl. Symposium on Computer Architecture*, 2009.

[15] M. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proc. Intl. Symposium on Computer Architecture*, 2009.

[16] H. Akinaga and H. Shima, "Resistive random access memory (ReRAM) based on metal oxides," *Proceedings of the IEEE*, 2010.

[17] C. Xu, D. Niu, N. Muralimanohar, N. P. Jouppi, and Y. Xie, "Understanding the trade-offs in multi-level cell ReRAM memory design," in *Proc. Design Automation Conference*, 2013.

[18] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating STT-RAM as an energy-efficient main memory alternative," in *Proc. Intl. Symposium on Performance Analysis of Systems and Software*, 2013.

[19] https://www.micron.com/about/our-innovation/3d-xpoint-technology.

[20] L. Jiang *et al.*, "A low power and reliable charge pump design for phase change memories," in *Proc. Intl. Symposium on Computer Architecture*, 2014.

[21] P. M. Palangappa and K. Mohanram, "CompEx: Compression-expansion coding for energy, latency, and lifetime improvements in MLC/TLC NVM," in *Proc. Intl. Symposium on High Performance Computer Architecture*, 2016.

[22] D. Niu *et al.*, "Low power multi-level cell resistive memory design with incomplete data mapping," in *Proc. Intl. Conference on Computer Design*, 2013.

[23] J. L. Henning, "SPEC CPU2006 benchmark descriptions," in *ACM SIGARCH Computer Architecture News*, 2006.

[24] V. Young, P. J. Nair, and M. K. Qureshi, "DEUCE: Write-efficient encryption for non-volatile memories," in *Proc. Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[25] S. Swami, J. Rakshit, and K. Mohanram, "SECRET: Smartly EnCRypted energy EfficienT non-volatile memories," in *Proc. Design Automation Conference*, 2016.

[26] M. Poremba and Y. Xie, "NVMain: An architectural-level main memory simulator for emerging non-volatile memories," in *Proc. Computer Society Annual Symposium on VLSI*, 2012.

[27] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. Conference on Prog. Language Design and Implementation*, 2005.

[28] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSS: a full system simulator for multicore x86 CPUs," in *Proc. Design Automation Conference*, 2011.

[29] X. Zhuang, T. Zhang, and S. Pande, "HIDE: An infrastructure for efficiently protecting information leakage on the address bus," in *Proc. Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[30] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in *Proc. Network and Distributed System Security Symposium*, 2012.

[31] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song, "Phantom: Practical oblivious computation in a secure processor," in *Proc. Conference on Computer & Communications Security*, 2013.

[32] T. John, S. K. Haider, H. Omar, and M. van Dijk, "Connecting the dots: Privacy leakage via write-access patterns to the main memory," in *International Symposium on Hardware Oriented Security and Trust*, 2017.

[33] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *Journal of the ACM*, vol. 43, no. 3, 1996.

[34] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: an extremely simple oblivious RAM protocol," in *Proc. Conference on Computer & Communications Security*, 2013.

[35] X. Yu, S. K. Haider, L. Ren, C. Fletcher, A. Kwon, M. van Dijk, and S. Devadas, "Proram: Dynamic prefetcher for oblivious RAM," in *Proc. Intl. Symposium on Computer Architecture*, 2015.

[36] L. Ren, X. Yu, C. W. Fletcher, M. Van Dijk, and S. Devadas, "Design space exploration and optimization of Path Oblivious Ram in secure processors," *Proc. International Symposium on Computer Architecture*, 2013.

[37] C. W. Fletcher, L. Ren, A. Kwon, M. van Dijk, and S. Devadas, "Freecursive oram: [Nearly] free recursion and integrity verification for position-based oblivious ram," in *Proc. Intl.*

Conference on Architectural Support for Programming Languages and Operating Systems, 2015.

[38] X. Zhang *et al.*, "Fork Path: Improving efficiency of ORAM by removing redundant memory accesses," in *Proc. MICRO*, 2015.

[39] R. Wang, Y. Zhang, and J. Yang, "Cooperative Path-ORAM for effective memory bandwidth sharing in server settings," in *Proc. Intl. Symposium on igh Performance Computer Architecture*, 2017.

[40] J. Rakshit and K. Mohanram, "LEO: Low overhead encryption ORAM for non-volatile memories," *IEEE Computer Architecture Letters*, vol. 17, no. 2, 2018.

[41] C. W. Fletcher *et al.*, "A low-latency, low-area hardware oblivious ram controller," in *Proc. International Symposium on Field-Programmable Custom Computing Machines*, 2015.

[42] B. C. Lee *et al.*, "Architecting phase change memory as a scalable DRAM alternative," in *Proc. Intl. Symposium on Computer Architecture*, 2009.

[43] C. Xu *et al.*, "Understanding the trade-offs in multi-level cell ReRAM memory design," in *Proc. Design Automation Conference*, 2013.

[44] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, *et al.*, "The architecture of the DIVA processing-in-memory chip," in *Proc. International Conference on Supercomputing*, 2002.

[45] G. H. Loh, "3D-stacked memory architectures for multi-core processors," in *Proc. International Symposium on Computer Architecture*, 2008.

[46] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: Throughput-oriented programmable processing in memory," in *Proc. International Symposium on High-performance Parallel and Distributed Computing*, 2014.

[47] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *Proc. International Symposium on Computer Architecture*, 2015.

[48] M. Radulovic, D. Zivanovic, D. Ruiz, B. R. de Supinski, S. A. McKee, P. Radojković, and E. Ayguadé, "Another trip to the wall: How much will stacked dram benefit hpc?," in *Proc. International Symposium on Memory Systems*, 2015.

[49] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," *Proc. International Symposium on Computer Architecture*, 2016.

[50] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *Proc. International Symposium on Computer Architecture*, 2016.

[51] H. Asghari-Moghaddam, Y. H. Son, J. H. Ahn, and N. S. Kim, "Chameleon: Versatile and practical near-dram acceleration architecture for large memory systems," in *Proc. International Symposium on Microarchitecture*, 2016.

[52] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, "Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation," in *Proc. International Conference on Computer Design*, 2016.

[53] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, K. Hsieh, K. T. Malladi, H. Zheng, and O. Mutlu, "LazyPIM: An efficient cache coherence mechanism for processing-in-memory," *IEEE Computer Architecture Letters*, vol. 16, no. 1, pp. 46–50, 2017.

[54] R. Hadidi, L. Nai, H. Kim, and H. Kim, "CAIRO: A compiler-assisted technique for enabling instruction-level offloading of processing-in-memory," *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 4, p. 48, 2017.

[55] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, *et al.*, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *Proc. Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[56] S. Ghose, K. Hsieh, A. Boroumand, R. Ausavarungnirun, and O. Mutlu, "Enabling the adoption of processing-in-memory: Challenges, mechanisms, future research directions," *arXiv preprint arXiv:1802.00320*, 2018.

[57] J. S. Kim, D. S. Cali, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, "GRIM-Filter: Fast seed location filtering in dna read mapping using processing-in-memory technologies," *BMC Genomics*, vol. 19, no. 2, p. 89, 2018.

[58] C. Gentry, *A fully homomorphic encryption scheme*. Stanford University, 2009.

[59] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, and M. K. Qureshi, "SYNERGY: Rethinking secure-memory design for error-correcting memories," in *Proc. International Symposium on High Performance Computer Architecture*, 2018.

[60] A. R. Alameldeen and D. A. Wood, "Frequent pattern compression: A significance-based compression scheme for L2 caches," *Dept. Comp. Scie., Univ. Wisconsin-Madison, Tech. Rep*, vol. 1500, 2004.

[61] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *Proc. Intl. Symposium on Computer Architecture*, 2009.

[62] N. H. Seong, D. H. Woo, V. Srinivasan, J. A. Rivers, and H.-H. S. Lee, "SAFER: Stuck-at-fault error recovery for memories," in *Proc. International Symposium on Microarchitecture*, 2010.

[63] D. H. Yoon, N. Muralimanohar, J. Chang, P. Ranganathan, N. P. Jouppi, and M. Erez, "FREE-p: Protecting non-volatile memory against both hard and soft errors," in *Proc. International Symposium on High Performance Computer Architecture*, 2011.

[64] A. P. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. Mossé, "Increasing PCM main memory lifetime," in *Proc. Conference on Design, Automation and Test in Europe*, 2010.

[65] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-Delta-Immediate compression: Practical data compression for on-chip caches," in *Proc. International Conference on Parallel Architectures and Compilation Techniques*, 2012.

[66] R. Melhem, R. Maddah, and S. Cho, "RDIS: A recursively defined invertible set scheme to tolerate multiple stuck-at faults in resistive memory," in *Proc. International Conference on Dependable Systems and Networks*, 2012.

[67] A. N. Jacobvitz, R. Calderbank, and D. J. Sorin, "Coset coding to extend the lifetime of memory," in *Proc. International Symposium on High Performance Computer Architecture*, 2013.

[68] R. Azevedo, J. D. Davis, K. Strauss, P. Gopalan, M. Manasse, and S. Yekhanin, "Zombie Memory: Extending memory lifetime by reviving dead blocks," in *Proc. International Symposium on Computer Architecture*, 2013.

[69] P. M. Palangappa and K. Mohanram, "Flip-Mirror-Rotate: An architecture for bit-write reduction and wear leveling in non-volatile memories," in *Proc. Great Lakes Symposium on VLSI*, 2015.

[70] A. Awad, P. Manadhata, S. Haber, Y. Solihin, and W. Horne, "Silent Shredder: Zero-cost shredding for secure non-volatile main memory controllers," in *Proc. Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[71] A. J. Menezes *et al.*, *Handbook of applied cryptography*. CRC press, 1996.

[72] B. D. Yang *et al.*, "A low power phase change random access memory using a data-comparison write scheme," in *Proc. Intl. Symposium on Circuits and Systems*, 2007.

[73] S. Cho and H. Lee, "Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance," in *Proc. Intl. Symposium on Microarchitecture*, 2009.

[74] S. K. Haider, *Architectural Primitives for Secure Computation Platforms*. PhD thesis, University of Connecticut, 2017.

[75] C. W. Fletcher, L. Ren, X. Yu, M. Van Dijk, O. Khan, and S. Devadas, "Suppressing the oblivious ram timing channel while making information leakage and program efficiency

trade-offs," in *Proc. International Symposium on High Performance Computer Architecture*, 2014.

[76] L. Ren, C. W. Fletcher, X. Yu, M. Van Dijk, and S. Devadas, "Integrity verification for Path Oblivious-Ram," in *Proc. High Performance Extreme Computing Conference*, 2013.

[77] S. K. Haider and M. van Dijk, "Flat ORAM: A simplified write-only oblivious ram construction for secure processor architectures," *arXiv preprint arXiv:1611.01571*, 2016.

[78] "Announcing the advanced encryption standard (AES)," *Federal Information Processing Standards Publication*, vol. 197, pp. 1–51, 2001.

[79] http://www.hpl.hp.com/research/cacti/.

[80] T. Kgil *et al.*, "ChipLock: Support for secure microarchitectures," in *Proc. Workshop on Architectural Support for Security and Anti-Virus*, 2005.

[81] B. Gassend, G. E. Suh, D. Clarke, M. Van Dijk, and S. Devadas, "Caches and hash trees for efficient memory integrity verification," in *Proc. Intl. Symposium on High Performance Computer Architecture*, 2003.

[82] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," in *Proc. Intl. Symposium on Computer Architecture*, 2006.

[83] D. Champagne *et al.*, "The reduced address space (RAS) for application memory authentication," in *Proc. Intl. Conference on Info. Security*, 2008.

[84] N. Chatterjee *et al.*, "Staged reads: Mitigating the impact of dram writes on dram reads," in *Proc. HPCA*, 2012.

[85] H.-Y. Cheng *et al.*, "Adaptive Burst-Writes (ABW): Memory requests scheduling to reduce write-induced interference," *Transactions on Design Automation of Electronic Systems*, vol. 21, no. 1, 2015.

[86] K. K.-W. Chang *et al.*, "Improving DRAM performance by parallelizing refreshes with accesses," in *Proc. International Symposium on High Performance Computer Architecture*, 2014.

[87] L. Ren, X. Yu, C. W. Fletcher, M. Van Dijk, and S. Devadas, "Design space exploration and optimization of path oblivious ram in secure processors," in *Proc. Intl. Symposium on Computer Architecture*, 2013.

[88] P. Rosenfeld *et al.*, "DRAMSim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, vol. 10, no. 1, 2011.

[89] "Micron DDR3 SDRAM datasheet, part MT41J512M8," *Micron Technology Inc.*, 2009.

[90] J. Kong and H. Zhou, "Improving privacy and lifetime of pcm-based main memory," in *Proc. International Conference on Dependable Systems and Networks*, 2010.

[91] Y. Choi *et al.*, "A 20nm 1.8 V 8Gb PRAM with 40MB/s program bandwidth," in *Proc. ISSCC*, 2012.

[92] A. Awad, Y. Wang, D. Shands, and Y. Solihin, "ObfusMem: A low-overhead access obfuscation for trusted memories," in *Proc. Intl. Symposium on Computer Architecture*, 2017.

[93] T. M. John, S. K. Haider, H. Omar, and M. Van Dijk, "Connecting the dots: Privacy leakage via write-access patterns to the main memory," *IEEE Trans. on Dependable and Secure Computing*, 2017.

[94] V. Costan and S. Devadas, "Intel sgx explained," *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016.

[95] https://www.arm.com/products/security-on-arm/trustzone.

[96] S. Sasy, S. Gorbunov, and C. Fletcher, "ZeroTrace: Oblivious memory primitives from Intel SGX," in *Proc. Symposium on Network and Distributed System Security)*, 2017.

[97] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, "AEGIS: Architecture for tamper-evident and tamper-resistant processing," in *Proc. International Conference on Supercomputing*, 2003.

[98] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proc. Intl. Symposium on Computer Architecture*, 2009.

[99] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: A hybrid PRAM and DRAM main memory system," in *Proc. Design Automation Conference*, 2009.

[100] A. P. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. Mossé, "Increasing pcm main memory lifetime," in *Proc. Conference on Design, Automation and Test in Europe*, pp. 914–919, European Design and Automation Association, 2010.

[101] H. G. Lee, S. Baek, C. Nicopoulos, and J. Kim, "An energy-and performance-aware DRAM cache architecture for hybrid DRAM/PCM main memory systems," in *Proc. Intl. Conference on Computer Design*, 2011.

[102] H. Yoon, J. Meza, R. Ausavarungnirun, R. A. Harding, and O. Mutlu, "Row buffer locality aware caching policies for hybrid memories," in *Proc. Intl. Conference on Computer Design*, 2012.

[103] T. J. Ham, B. K. Chelepalli, N. Xue, and B. C. Lee, "Disintegrated control for energy-efficient and heterogeneous memory systems," in *Proc. Intl. Symposium on High Performance Computer Architecture*, 2013.

[104] S. Lee, H. Bahn, and S. H. Noh, "CLOCK-DWF: A write-history-aware page replacement algorithm for hybrid PCM and DRAM memory architectures," *IEEE Trans. on Computers*, vol. 63, no. 9, 2014.

[105] W. Wei, D. Jiang, S. A. McKee, J. Xiong, and M. Chen, "Exploiting program semantics to place data in hybrid memory," in *Proc. Intl. Conference on Parallel Architecture and Compilationn*, 2015.

[106] W. Wei, D. Jiang, J. Xiong, and M. Chen, "HAP: Hybrid-memory-aware partition in shared last-level cache," *ACM Trans. on Architecture and Code Optimization*, vol. 14, no. 3, p. 24, 2017.

[107] Y. Zhou, R. Alagappan, A. Memaripour, and A. B. D. Wentzlaff, "HNVM: Hybrid nvm enabled datacenter design and optimization," *Microsoft, Microsoft Research, Tech. Rep. MSR-TR-2017-8*, 2017.

[108] H. Sun, J. Liu, R. S. Anigundi, N. Zheng, J.-Q. Lu, K. Rose, and T. Zhang, "3D DRAM design and application to 3D multicore systems," *IEEE Design & Test of Computers*, vol. 26, no. 5, 2009.

[109] T. Zhang, C. Xu, K. Chen, G. Sun, and Y. Xie, "3D-SWIFT: A high-performance 3D-stacked wide IO DRAM," in *Proc. Great Lakes Symposium on VLSI*, 2014.

[110] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu, "Simultaneous multi-layer access: Improving 3D-stacked memory bandwidth at low cost," *ACM Trans. on Architecture and Code Optimization*, vol. 12, no. 4, p. 63, 2016.

[111] C. C. Chou, *Architecting high-performance, efficient, and scalable heterogeneous memory systems with 3D-DRAM*. PhD thesis, Georgia Institute of Technology, 2017.

[112] Y. Eckert, N. Jayasena, and G. H. Loh, "Thermal feasibility of die-stacked processing in memory," in *Proc. Workshop on Near-Data Processing*, 2014.

[113] S. Swami, J. Rakshit, and K. Mohanram, "STASH: Security architecture for smart hybrid memories," in *Proc. Design Automation Conference*, 2018.

[114] N. Fazio, A. R. Nicolosi, and I. M. Perera, "Oblivious group storage," in *Proc. Conference on Computer and Communications Security*, 2013.

[115] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang, "Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward," in *Proc. Conference on Computer and Communications Security*, 2015.

[116] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. Van Dijk, and S. Devadas, "Constants count: Practical improvements to Oblivious RAM," in *Proc. USENIX Security Symposium*, 2015.

[117] J. L. Henning, “Spec cpu2006 memory footprint,” *SIGARCH Computer. Architecture News*, vol. 35, no. 1, pp. 84–89, 2007.

[118] T. Moataz, T. Mayberry, E.-O. Blass, and A. H. Chan, “Resizable tree-based Oblivious RAM,” in *Proc. International Conference on Financial Cryptography and Data Security*, 2015.

[119] M. K. Qureshi, “Pay-As-You-Go: Low-overhead hard-error correction for phase change memories,” in *Proc. International Symposium on Microarchitecture*, 2011.

[120] S. Swami, P. M. Palangappa, and K. Mohanram, “ECS: Error-correcting strings for lifetime improvements in nonvolatile memories,” *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 4, p. 40, 2017.