Abstract of "Towards a Practical Secure Memory for Modern Deployment" by Samuel Thomas, Ph.D., Brown University. April 28, 2025.

Processors rely on memory devices to store data that cannot does not fit in on-chip components, such as registers and caches. In doing so, an implicit assumption is made that data that is loaded on chip from memory is consistent with the state that was stored. This primitive is critical for security and privacy; user data and program state alike are stored in memory, so its corruption can lead to dangerous behaviors such as malicious execution or data leakage. Unfortunately, memory devices are subject to a variety of well known attacks that allow for this corruption.

Given the potential vulnerability of memory devices, the study *secure memory*, which describes how processor can reason about the storage of data in vulnerable memory devices, is of particular interest. Doing so explicitly enforces the implicit assumptions by extending the memory controller logic to maintain metadata associated with the data stored in memory. This is a valuable primitive as it allows processors to reason about the privacy and integrity of stored data. With that said, such a protocol comes with several limitations to its practicality: ① implementing secure memory comes at the cost of runtime performance; ② security metadata requires a significant space to store; and ③ primitive definitions in secure memory limit its adaptability to emerging memory technologies. As a result, more recent versions of secure processors seldom implement a comprehensive secure memory protocol.

In order to make secure memory a more palatable feature for commodity secure hardware, it is necessary to develop an in-depth understanding of the secure memory protocol. This dissertation first contributes a robust study of secure memory entailing a detailed qualitative and quantitative description of its mechanisms, the translation of this theoretical protocol to practice, and the state-of-the-art optimizations to the protocol. Doing so will classify the overheads of each of its components and highlight the impact of the various design decisions that lead to the current secure memory landscape.

This dissertation will introduce four novel adaptations of the secure memory protocol: two of which extend its baseline approach and two of which target its adaptability to emerging technologies. It will first introduce *Cordelia*, a modification to the secure memory protocol that benefits runtime performance. This protocol is robust to the trend of increasingly memory-intensive applications. Furthermore, it proposes the *Baobab Merkle Tree* to alleviate the spatial overhead pressure imposed by secure memory metadata. Each of these protocols leverage insights concerning runtime behaviors of the application and metadata to optimize each of these ends. Afterwards, this dissertation explores the impact of two emerging memory technologies on the secure memory protocol. It will first propose *A Midsummer Night's Tree*, which describes an adaptation of secure memory for non-volatile memories that explores the trade-off space imposed by these devices in-depth. Then it will propose *CAPULET* and describe how emerging disaggregated memories can be leveraged to improve the existing optimizations to secure memory. From each of these ends, this dissertation will formalize the problems that these emerging technologies impose on the practicality of secure memory.

Towards a Practical Secure Memory for Modern Deployment

by
Samuel Thomas

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
April 28, 2025

This dissertation by Samuel Thomas is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date 4/15/2025 DocuSigned by: Iris Bahar 9374B6CBB3C6443...

Dr. R. Iris Bahar, Director

Recommended to the Graduate Council

Date 4/15/2025 Signed by: Malte Schwarzkopf 1EB6208B145D49E...

Dr. Malte Schwarzkopf, Reader

Date 4/15/2025 DocuSigned by: Maurice Herlihy 9E6086620166498...

Dr. Maurice Herlihy, Reader

Date 4/15/2025 DocuSigned by: Tamara Lehman 827EAE87A2A74C7...

Dr. Tamara Lehman, Reader

Approved by the Graduate Council

Date _____

Dean of the Graduate School

Vita

Samuel Thomas was born in Boston, Massachusetts. Prior to Brown, he completed his BSc at Davidson College, where he graduated with a major in Political Science and with High Honors in Computer Science. He received his ScM in Computer Science at Brown en route to his Doctoral degree.

Samuel's research has been published at ACM ASPLOS, IEEE Computer Architecture Letters, and IEEE HPEC. He has multiple works under submission, which are described in this dissertation. His has given an invited talk at EMERALD, a workshop co-located with SPAA, to speak about his work in the context of parallel applications. He holds a teaching certificate from Brown University's Sheridan Teaching Center, and he has mentored undergraduate and early graduate students towards publication at the YArch workshop (co-located with ASPLOS).

Preface

This thesis bridges the gap between the theoretical literature in secure memory with the problems towards its practical realization and deployment. Ideally, secure memory is an established feature of any secure hardware that is easily accessible to end-users. Unfortunately, secure memory is presently far from this ideal. This thesis argues that the reason for this is a three-fold limitation of its theoretical conception: ① secure memory is performance limiting, particularly in terms of bandwidth consumed for metadata fetches; ② secure memory metadata is storage-intensive, especially if optimizations require even further metadata; and ③ secure memory does not abstract well to emerging memory technologies.

To begin to address these limitations, this dissertation first explores and profiles the design decisions culminating in the first (since rescinded) commodity release of secure memory. In doing so, it provides context of the potential benefit and room for optimization across various design directions. From here, the thesis goes into several proposed secure memory architectures that address the argued limitations. In particular, it proposes ① *Cordelia* to address its runtime performance and bandwidth overhead, ② the *Baobab Merkle Tree* to address the metadata storage overhead of secure memory, ③ *A Midsummer Night's Tree* to address the application of secure memory to non-volatile memories, and ④ *CAPULET* to address the application of secure memory to disaggregated memories.

Acknowledgments

This dissertation would not be possible without significant intellectual and personal support. I am humbled, honored, and grateful to all of the people in my life who have been by my side in the process.

First and foremost, I am extraordinarily grateful to my advisor, **Iris Bahar**. Developing my research infrastructure would have been impossible without Iris' guidance about navigating research, teaching, and mentorship. I am forever indebted to the commitment you've entrusted in my research. In a professional context, thank you for giving me the freedom to stumble when learning how to find my way as an academic but never letting me fall. On a personal level, thank you for building such a wonderful community (in spite of a global pandemic!) and welcoming me into your group at Brown, and thank you for continuing to make me feel welcome in my time in Colorado.

I would be remise not to express my deepest gratitude for the rest of my committee as well: **Malte Schwarzkopf**, **Maurice Herlihy**, and **Tamara Lehman**. I am so grateful for your assistance and feedback in the process of formalizing the work throughout this dissertation. Beyond the dissertation, thank you for challenging me throughout my PhD. I will leave Brown beaming with pride over my research, but this would not be possible without the confidence in its motivation, intellectual and technical merits, and communication. Thank you for engaging with me on each of these ends, for actively listening to my responses, and for refining my intuition.

I also feel extraordinarily grateful to have, in my opinion, the best network of collaborators (and I refuse to hear dissenting opinions). Without fail, whenever any of your names come up in a conversation at a conference or workshop, the other people say "Oh, they are the best!" and I cannot agree more. Outside of my committee members, I want to especially thank **Tali Moreshet**, **Joseph Izraelevitz**, and **Erez Petrank** for being exceptional faculty collaborators. Tali, thank you for your consistent commitment to help refine and understand narratives with whatever wild and crazy idea I presented in research meetings. Joe, thank you for teaching me how to have honest conversations about navigating the intellectual and professional status of a research project. Erez, thank you for being such a strong example of setting and consistently reaching a high standard of research. I know these examples only scratch the surface of what you all have taught me, and I am so grateful for feeling so supported every interaction I've had with all of you. I've also been tremendously fortunate to work with outstanding graduate student collaborators: **Jiwon Choe**, **Ofir Gordon**, **Kidus Workneh**, **Ange-Thierry Ishimwe**, **Zack McKevitt**, **Phaedra Curlin**, **Gal Sela**, **Hunter Thompson**, **Will Buziak**, and **Zach Moolman**. I am so grateful for each of your commitment to the work we did together, and I am so impressed by all of your research and accomplishments. Finally, I have been honored to work with so many fantastic undergraduate students: **Jac McCarty**, **Chia Jen Cheng**, **Adam Richling**, **Kasper Selgem**, **Neil Ramaswamy**, **Aidan Nowakowski**, **Eliott Dinfotan**,

Hammad Izhar, **Suhana Zeutzius**, **Fadi Kidess**, and **Connor Bremner**. It has been a privilege getting to share what I know with you all. I am forever grateful for your efforts, and I hope you all took as much away from the experience of working together as I did.

Outside of research, my PhD would not have been possible without such strong positive mentorship and community. I would like especially to thank **Tamara Lehman** for acting as essentially a second advisor in so many ways throughout my PhD. Thank you for your guidance on research and for always making sure that I am set up for success. I feel so lucky to be included as a part of your group with Joe at Boulder. In addition, I would like to express my deepest gratitude to **Milda Zizyte** for your mentorship. Getting to help you design Computer Architecture was such a privilege. I learned so much about pedagogy from you, and am forever grateful for the freedom and support you showed me before and throughout the term. I would also like to thank **Jiwon Choe**, **Jasmine Liu**, and **Semanti Basu** for being such great lab mates. To **Franco Solleza**, **Roma Patel**, **Ellie Eng**, and **Justus Adam**: thank you for always being down to go on an adventure. When I think back on my PhD, my memories will be equally split between the CIT and the mountain air of New Hampshire and Maine. Thank you for hiking, skiing, camping, and singing with me (in addition to the occasional shop talk) as well as always being down for a coffee run. To the whole of A-Staff, thank you for making all of the logistics of my degree and every other department function so smooth and seamless. In particular, thank you to **Lauren Clarke** and **John Tracey-Ursprung** for answering my plethora of questions across my countless random drop-ins.

I wear my pride for my undergraduate alma mater, Davidson College on my sleeve. In large part, this is because of the amazing people that I met there and the importance of their support throughout my PhD cannot be understated. To **Hammurabi Mendes**: I feel so lucky to get to brag about my "outstanding undergraduate advisor" at every opportunity. I am forever indebted to the love of computing that you bestowed in me and for the honest, genuine guidance at every major stage of my life since my sophomore year at Davidson. Thank you from the bottom of my heart for your consistent support and friendship. I wake up every day grateful for my best friends, **Alex Hazan** and **Jake Clary**. Thank you both for your friendship. For reasons too many to enumerate in this document, this degree would have been impossible without you guys. I love you both. To **Caroline Evans**, thank you for your academic perspectives on all sorts of matters and your consistent friendship.

Finally, I would like to thank my family. To my **Mom**, **Dad**, my siblings **Jon** and **Em**, **Nana** and **Zada**: thank you all for supporting me in so many ways over these five years. I am so grateful for all of our phone calls and support when things were tough. I love you all so much, and I could not have done this without you. And a deep thank you to my **Grandma**, who I would complain to in all of the difficult moments. She would always tell me, "What did you expect? You did choose to get a PhD." I love you, and I miss you every day.

Dedication

To my Grandma, and to choosing to get a PhD.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

For the for several decades, researchers have asked how a processor can safely execute a memory-dependent program if the memory itself is untrusted [160, 172, 94, 93, 159, 248, 299, 236, 246, 298, 222, 81, 221, 80]. Suppose a processing element was unable to store all of a program's data in the processor state (i.e., registers, cache hierarchy, etc). Therefore, that data must be stored in a "remote" storage device (i.e., in a memory device off of the processor chip). Seeing as the device is remote,it is natural to ask what can happen if the device can be tampered or manipulated. What happens if an adversary can maliciously modify the contents of data? Should the processor trust any data responses from the remote device? If memory should not be trusted, then there is a need for secure memory.

At its core, this theoretical proposition is a function of fundamental implicit assumptions that characterize the interaction of heterogeneous components in a computer's architecture. A processor accesses a memory device through a simple load/store interface. Whenever there is some data that must be stored externally, the processor issues a "store" request to the memory device with the data and its address. When that data is later needed by the program, the processor issues a "load" request to the memory device for the data address. The simplicity of this interface allows for an elegant interaction between the processor and the memory device, but it also limits the processor's ability to reason about the state of the program's data while it was stored. That is, the processor *assumes* that the data's state is safely maintained under all circumstances. But is this assumption reasonable? Should a processor trust external devices to uphold these assumptions? In the context of memory, this big question can be boiled down to two key properties about the maintenance of stored data. While the data was stored, is its state private as the processor, application, and developer expect? Furthermore, was the state of the data preserved while it was stored?

As it turns out, memory devices are subject to a broad set of physical vulnerabilities [305, 188, 175, 138, 171]. Among these vulnerabilities, an adversary can leak the contents of data in memory [305, 218] or corrupt its contents [303, 190, 168, 120] through a variety of means (via physical tampering or software). As a result, main memory serves as a powerful, natural attack surface [142, 240, 295] for an attacker to target a victim and gives credence to the premise that processors should not implicitly trust memory devices.

For this reason, secure memory has been implemented in a variety of contexts in commodity devices [13, 130, 209, 14]. These works build on practical implementations of early literature such that developers can

deploy security sensitive applications in dedicated secure hardware (enclaves) in which the *hardware* guarantees certain protections to application, including the vulnerabilities of the memory devices. Explicitly maintaining guarantees of data privacy and integrity in hardware is an arduous process for large scale or general purpose processing, and it would impose significant limitations on application performance and memory device design. For this reason, manufacturers place artificial limitations on types of workloads that can use this hardware. Intel has restricted developers to either limited enclave size to MBs of capacity [13] or relaxed protections for total memory protection [130, 112, 169], whereas Apple has limited the application of their secure hardware to a small set of workloads (e.g., passocde change, enabling/disabling login preferences, adding/removing payment information, erasing all content/settings) rather than for general purpose utilization [14]. Even in the restricted deployment, secure memory has been demonstrated to face practical performance limitations [312, 302, 95, 71, 115].

It is crucial to improve secure memory so that it is practical for these commodity deployments. The limited scale of its deployment is suboptimal as there are a natural applications that would benefit from secure memory but are outside of its currently deployed restricted accessibility. Consider running an application in a remote, shared cloud. These workloads are generally large-scale deployments (otherwise they would be run locally), and the breadth of application types in the cloud is essentially unlimited. However, an application in the public cloud is shared with potentially malicious guests/administrators and the developer has little to no control over the software platforms (i.e., library or operating system versions) much less who has physical device access.

This is the state of the world the state of the world: memory devices are subject to dangerous vulnerabilities with known defenses, but the defenses are largely inaccessible for developers of commodity deployment. As a result, these protocols may as well not exist as developers remain subject to the vulnerabilities of memory devices. The work in this dissertation emphasizes the modernity of the secure memory problem: the protections guaranteed by this literature are important, therefore new work in secure memory must consider its impracticality as a first-order consideration so that secure memory may be realized in commodity deployment.

1.2 Understanding the Problem

When executing a program, the processor and memory engage in an implicit handshake agreement concerning the state of the data. To store data, the processing element asks memory to maintain its state to an address in memory. Similarly, when the processing element later requests the data from that address, it assumes that the memory device will return the data in its expected state. With that said, this model of interactions between processing elements and memory devices hides implicit assumptions concerning the state of data between when the initial store request and the subsequent load request. For instance, it assumes that data is kept private according to the behavior of the application and that its state is free of corruption.

Unfortunately, these assumptions are impossible to guarantee without explicit enforcement. For instance, memory devices are subject to a host of leakage-based vulnerabilities [102, 100, 285, 300] and corruption-based vulnerabilities [303, 190, 168, 131, 120]. To assume that data is kept private from unintended access assumes that only a single process uses the computing platform (i.e., processing elements and memory system) at a time, which only describes rare computational instances [35, 97, 203, 15, 283]. Otherwise, memory may be accessed via other processes sharing the processor, accelerators on the platform, I/O, etc. Worse, memory

devices cannot guarantee the state of stored data. Memory devices are real hardware subject to glitches, imperfections, the environment, etc. To be concrete, most deployed modern memories are dynamic random access memories (DRAMs). A DRAM cell (i.e., a single bit of data) is composed of a single transistor and a single capacitor. The capacitor is kept at a positive or negative charge to represent one or zero, and the transistor emits current into the cell to update the state during writes or discharge the state during reads. However, this integrated circuitry is known to be unreliable. For this reason, many DRAM-based main memories typically store error correction codes (ECC) [106] in memory as metadata alongside data to correct single bit faults in a byte. At the same time, lightweight ECC mechanisms are insufficient to defend against more comprehensive corruptions. Thus, memory is a natural target for an adversary to leak sensitive information and/or tamper with a victim's essential program state to trigger malicious behavior. As such, vigorous memory protection of physical memory devices is essential to ensure the safety of the memory system.

Secure memory may be implemented to protect against these vulnerabilities. Generally, secure memory describes a protocol in which the privacy and integrity of data in memory are explicitly protected [247]. This protection is implemented an extension to the logic in the memory controller device, and guarantees that the state of any data used by a processing element was not revealed nor maliciously modified while residing in memory. To do so, data stored in a secure memory has some associated metadata. In particular, secure memory encrypts data to explicitly guarantee the privacy of data and maintains some metadata for authentications of the data's state to explicitly guarantee its integrity. This metadata takes the form of an integrity tree (e.g., a Merkle tree [181]) over the data state against which any fetched data must be authenticated. Thus, the task of maintaining data in a secure memory is equally the task of maintaining its associated metadata: fetching some data requires fetching its associated metadata so that the data can be authenticated. Updating data requires also updating the associated metadata for future authentications.

Secure memory faces a natural performance limitation. Encrypting/decrypting data and authenticating its state adds latency to the critical path of its fetch from memory and maintaining its associated metadata consumes some of the memory device's available bandwidth. Both of these features of a secure memory can be performance limiting for workloads that assume low memory latency and high bandwidth. Furthermore, storing this metadata requires reserving space in the memory device that cannot otherwise be used for application data. As a result, applications are more likely to need to store data in slower storage devices with a secure memory.

1.3 Challenges and Goals

Several works from academic and industry circles have studied the deployment of secure memory in commodity devices [130, 112, 169]. They study the commodity deployment of SGXv1 [13, 99] which in many ways reflects the convergence of literature describing how to securely execute a program with an untrusted memory. There are three key takeaways from these works: ① secure memory imposes a consequential performance overhead on memory-bound applications; ② the spatial overhead of secure memory metadata is a significant challenge; and ③ using emerging memory technologies and architectures impose significant semantic differences to which secure memory does not naturally generalize. The work in this dissertation explores each of these challenges explicitly.

1.3.1 Performance

Challenge In order to provide confidentiality and authenticate the integrity of data stored in memory, secure memory needs to fetch and maintain metadata associated with data stored in the memory device. As a result, any additional work that a secure memory needs to perform to make these guarantees will come at a performance cost relative to the insecure solution. The performance overhead attributed to maintaining secure memory metadata has been well studied, but the solutions do not extend to nor consider the demands that modern workloads make of memory devices.

Goal When considering the performance overhead of secure memory, the work in this dissertation considers how the application behavior limits the efficacy of any optimization. As a result, this dissertation considers the application as a first-order consideration when designing optimized secure memory architectures to account for the otherwise limiting performance.

1.3.2 Spatial Overhead

Challenge Secure memory requires significant space both in memory and on-chip. The metadata associated with secure memory comes at a steep storage overhead. Over 14% of the capacity of a memory device must be allocated to maintain the metadata associated with secure memory. Each byte of a memory device allocated to secure memory metadata comes at the expense of a byte to be used by an application. In addition, secure memory requires a logical extension to the on-chip memory controller device, which occupies on-chip area. Further extensions to this logic will demand an even greater area requirement to the already limited space on-chip. For example, reducing the size of the last-level cache (LLC) to account for increased memory controller area will result in fewer instructions being handled by the cache hierarchy and more requests being made of main memory.

Goal The work in this dissertation takes spatial overhead into account. The proposed secure memory architectures either explicitly aim to reduce the memory storage requirement of secure memory or discuss their implications on storage overhead. Furthermore, work in this dissertation describes complexity as a first-order consideration.

1.3.3 Emerging Memory

Challenge Main memory is not a stagnant phenomenon. Commodity deployments of new technologies and architectures has become mainstream by way of persistent non-volatile memories (NVMs) [297, 127, 293, 54, 49, 291] and disaggregated memories (e.g., NUMA [145], CXL [67], NVLink [200], UALink [268]), etc. These devices and architectures possess properties that impose new programming models for correct execution. A correct implementation of secure memory similarly must adapt to these new semantics. For example, persistent memory programming requires ensuring the crash consistency of application data [223]. Similarly, the crash consistency of secure memory metadata must be ensured for correct execution [23]. On the other hand, the appropriate programming model for disaggregated memories has yet to be agreed upon [43, 69, 19, 296]. Similarly, the task of securing multiple memory devices who may be accessed by multiple hosts is poorly defined [10, 3, 85, 238].

Goal For both NVMs and disaggregated memories, this dissertation explicitly extends secure memory to account for the new semantics imposed by emerging memories. In particular, this dissertation proposes an architecture that explicitly considers the limitations that crash consistency impose on maintaining secure memory metadata. Furthermore, this dissertation explores the necessary assumptions to use disaggregated memory as an opportunity to optimize secure memory.

1.4 Thesis Statement

The thesis statement for this dissertation is as follows:

> To make secure memory practical for commodity deployment, secure memory should be extended to address performance, storage, and adaptability limitations as first-order design considerations.

1.5 Contributions

Towards the ends of producing secure memory that is suitable for commodity deployment, this dissertation explores secure memory from two key angles: ① how might securing an untrusted memory device be made more efficient for modern deployment, and ② how do emerging memories challenge preconceived secure memory semantics. Towards each of these ends, this dissertation makes the following contributions:

1.5.1 Improving Secure Memory

Cordelia A key component of ① is the fact that the integrity tree serves a performance bottleneck to modern applications. Thus, to address the performance limitations of a secure memory, this dissertation first proposes *Cordelia* (**under submission**), a Huffmanized Merkle Tree to authenticate memory accesses. In particular, Cordelia works from the observation that the central data structure in secure memory is designed agnostic to the context of application behavior. In many cases, this can be reconciled by the fact that the architecture, by way of the metadata cache, can help accelerate some recently accessed authentication paths. However, caches are not a panacea. Applications have made increasing demands of memory bandwidth through large scale processing of big data, and the utilization of caches with a small capacity (i.e., the metadata cache) struggle to keep up.

To this end, Cordelia accounts for the optimization promised by the metadata cache by constructing the integrity tree as a Huffman tree. As a result, the number of metadata accesses to fetch in order to authenticate some data is a function of how often that data has been accessed. That is, more frequently accessed data have shorter paths through the integrity tree.

Baobab Merkle Tree In addition to the performance bottleneck imposed by secure memory, maintaining secure memory metadata in memory comes at a significant spatial overhead. To this end, this dissertation proposes the *Baobab Merkle Tree* [261] that memoizes encryption counters on-chip to reduce the spatial overhead of integrity trees. Implementing secure memory often uses encryption counters as the basis of the integrity tree. If the encryption counters are memoized on-chip, then the basis of the integrity can be a reference to the encryption counter on-chip.

The Baobab Merkle Tree works from the insight that many encryption counters are similar values (i.e., zero or one) and therefore do not require extensive storage space in memory. As a result, the in memory reference can be a small number of bits to refer to the index in the memoization table. This dissertation demonstrates how guaranteeing the integrity of these references is equivalent to guaranteeing the integrity of encryption counters in a secure memory.

1.5.2 Securing Emerging Memories

A Midsummer Night's Tree Non-volatile memories (NVMs) impose new semantics on how processors and memories interact. In particular, applications must explicitly instruct processors to account for the crash consistency of data as it moves through the volatile and non-volatile storage regions. Similarly, the crash consistency of secure memory metadata must be accounted for when securing a NVM. However, doing so naïvely can further limit the performance of a secure memory or lead to long wait times when restoring the system after a loss of power. This dissertation proposes *A Midsummer Night's Tree* (AMNT) [260] to explicitly account for the crash consistency of secure memory.

AMNT considers various strategies as a fundamental component in designing the secure NVM. In particular, it tracks a single subtree in which a majority of accesses occur and runtime crash consistency costs are kept low. Outside of the subtree, which comprises the majority of addresses, there is no cost to securely recovery the memory device after a power loss. Tracking the fast subtree is achieved both in the architecture and by leveraging hardware-software co-design to keep the logical extensions to the memory controller lightweight.

CAPULET Disaggregated memories, such as CXL [67], have emerged as a technology in which multiple memory devices may be attached to one or more hosts via I/O. However, this deployment scenario raises important questions of how to secure data in a remote memory device that may be legitimately accessed by another legitimate but unknown host. This dissertation proposes *CAPULET* (**under submission**) to build efficient secure memory for these cases. In particular, CAPULET formalizes the candidate schemes for how metadata should be maintained safely across memory devices in a disaggregated memory system.

In addition, this dissertation makes the argument that secure memory can leverage the fabric of an interconnected disaggregated memory to optimize data authentication. Lightweight modifications to the communication protocol between hosts in a disaggregated memory can allow for the secure private communication of metadata between devices. Given this primitive, hosts that share a disaggregated memory system can coordinate the storage of secure memory metadata in the global set of metadata caches as a distributed system. As a result, underutilized caches in the distributed system of caches can dynamically lend space to hosts under strain to alleviate authentication bottlenecks elsewhere in the system.

1.6 Outline

The remainder of this dissertation is organized as follows:

Chapter 2 describes the relevant background concerning secure memory. In particular, it describes the threat model in the context of the vulnerabilities of memory devices and the attacks that leverage these vulnerabilities. From here, this chapter describes the protocol to guarantee the privacy and integrity of

data in an untrusted device. This discussion entails a description of the design decisions and trade-offs, an empirical study of each component, and a discussion of the literature that optimizes this component.

Chapter 3 describes Cordelia. To do so, this chapter starts with an intuition into the implications of secure memory on device bandwidth. This will motivate how our approach improves the performance of secure memory relative to the baseline protocol despite the additional latency incurred on an individual authentication. From here, this chapter will describe the design and performance implications of the approach. This section will conclude by describing the impact of Cordelia on the state of secure memory, the limitations of the approach, and the future areas for exploration born out of this study.

Chapter 4 describes the Baobab Merkle Tree. In particular, this section starts by describing the sources of spatial overhead due to secure memory metadata. From here, it presents the techniques applied to memoize secure memory metadata on-chip. This entails a description of the protocol, the performance implications of such a protocol, and the strategies to most effectively utilize such a protocol. This section will then describe the implications of this approach on runtime and storage overhead relative to the baseline approach. Finally, this chapter will describe limitations and future directions for exploration born out of this approach.

Chapter 5 describes A Midsummer Night's Tree. This protocol targets secure memory for NVMs, so this chapter starts by detailing the problem of maintaining crash consistency in secure memory metadata. From here, this chapter details the metadata crash consistency protocol and its implications. This chapter also describes how co-designing the physical memory allocation and reclamation procedure in the operating system can improve the underlying architecture within the scope of the threat model. This discussion provides the basis for the performance and recovery implications of the protocol. This chapter concludes with a discussion of the implications of the approach and areas for future exploration born out of this work.

Chapter 6 describes CAPULET. To do so, this chapter starts by conveying the challenges associated of deploying secure memory in the context of disaggregated memories. In particular, this chapter details various approaches for establishing trust in a system of disaggregated hosts and memories. From here, this chapter delves into how to configure memory controllers of a set of hosts and memories such that they may be able to efficiently protect the state of the underlying data and the challenges presented by implementing such a scheme naïvely. This chapter then overviews how to develop a cache pool of metadata caches by modifying the communication protocol between hosts and memory. This allows a discussion of how disaggregated memories can benefit the performance of secure memory. Finally, this chapter concludes with a discussion of the limitations of this approach and areas of future exploration in secure disaggregated memories.

Chapter 2

Background

2.1 Nomenclature

This chapter describes the secure memory protocol. That is, it explores how a processor can securely execute a program that stores data in an untrusted memory device. Generally, this protocol is implemented as a logical extension to the on-chip memory controller by way of a memory encryption engine (MEE). When fetching some data from memory, it is fetched at the "word" granularity, which refers to a 64-byte (64B) block of data. This chapter also refers to pages, which describe 4-kilobyte (4kB) chunks of data. By on-chip, this dissertation refers to the CPU(s), the register state, and the cache hierarchy as these are the devices in which data is stored. The memory controller describes a device that sits at the edge of the processor boundary (i.e., the interface between on-chip and off-chip) through which requests for the memory device are serviced. On-chip components comprise the trusted computing base (TCB).

The secure memory protocol describes a procedure in which data has its privacy guaranteed and its integrity authenticated. To keep data private, the MEE encrypts it prior to its storage in untrusted memory. Data is typically encrypted using counter-mode encryption (CME) in which each data has an associated counter as metadata. The integrity of data is guaranteed by maintaining two metadata fields: ① a message authentication code (MAC) and ② an integrity tree. MACs are typically hashes of the data state (referred to as hashed MACs, or HMACs).

The integrity tree is a tree of hashes over the encryption counters, referred to as a Bonsai Merkle Tree (BMT). Elements in the BMT are referred to as nodes and a node is defined as the hash of its associated children. Thus, authentication with a BMT requires computing the hash of the untrusted data and comparing it against the expected value (i.e., the parent node). Seeing as the parent node may be untrusted, this process is repeating recursively until it is authenticated against some trusted value (i.e., the on-chip BMT root). This sequence of nodes used to authenticate some data is referred to that data's "authentication path" and is sometimes referred as its "ancestry" through the BMT. To reduce the height of the tree, the BMT is 8-ary (i.e., each node has eight children) to increase its density.

To help optimize secure memory, the MEE employs a small cache for metadata fields (i.e., encryption counters, HMACs, BMT nodes). This cache is referred to as the "metadata cache." Seeing as this cache resides in the MEE (and in the on-chip memory controller by proxy), values in the metadata cache are considered trusted and can serve as "roots of trust" for authentication.

2.2 Establishing the Threat Model

2.2.1 Adversaries and Victims

The "A-B-C" attacker profile [38] characterizes attackers as conforming one of three potential profiles. ① An access seeking attacker wishes to gain login access on a platform to which they do not have access. For example, consider a locked mobile computing device that has been stolen by the adversary. ② A breaching attacker has guest access to a platform shared, and may wish to escalate their privileges or corrupt other processes co-located on the platform. An example of this may be a legitimate guest user of a cloud resource or guest application on a victim's local device (i.e., a webpage). ③ A conspiring attacker may both legitimately run root privileged software on the same platform as a victim and have physical access to the platform. Such an adversary may be the provider and operator of a cloud instance, and any user of the cloud system is a potential victim. Thus, an attacker may be able to execute software with various degrees of privilege and may have physical access to a device. Note, this classification aims to build an intuition of the potential adversary, but by is by no means comprehensive.

In the aforementioned profiles of attackers serve as adversaries for a variety of potential victims. In particular, an operating system or container may restrict the permissions granted to a breaching attacker, and a guest process may be the target of a conspiring attacker. To ensure the safety of a victim from memory vulnerabilities, the memory must be safe from illegitimate reads and writes of its contents. If an adversary has the ability to read the contents of memory, they may be able to read sensitive victim data (e.g., passwords, private end-user data, secret keys, proprietary program state, etc). Furthermore, if an adversary can write the contents of data, they may explicitly update their permissions [21, 76, 226], inject malware [244], or execute arbitrary malicious payloads [39].

2.2.2 Physical Vulnerabilities of Memory Devices

Attacks that maliciously utilize device properties to trigger unintended behaviors are considered *physical attacks*. These attacks may be performed in a variety of contexts. Main memory provides a load/store interface via the memory controller device [45]. Traffic to the memory controller may come from a variety of sources, and serves as a convenient backdoor for exploitation as a result. Furthermore, physical features of memory devices (i.e., data remanence [102], electromagnetic coupling [48, 184, 217], bridges between signal lines [18], and hot-carrier injections [55]) all serve as potential vulnerabilities for data leakage (read) and/or corruption (write).

Read. Seeing as legitimate accesses from the memory hierarchy may appear on the memory bus as last-level cache (LLC) misses or writebacks, an attacker with physical access to the device may inject traffic on the memory bus to emulate this behavior [116]. Legitimate memory requests may also come from the processor via a direct memory access interface (DMA) [21], the network interface card (NIC) [76], an I/O device [226, 27, 228], etc. These interfaces with the memory device are important for legitimate purposes such as remote debugging [243], forensic analysis [44], and/or legal interception of network traffic [206]. At the same time, these interfaces similarly allow malicious channel to leak data at an adversary-specified address. Assuming an adversary knows where some sensitive data is stored, they can use these mechanisms to leak it and achieve their malicious ends.

Suppose, however, that an adversary does not have prerequisite knowledge of where some sensitive data is located. To work around this, the attacker may want to leak all data in memory on which they can process offline. To achieve this, they may perform a cold-boot attack [102]. Cold-boot attacks leverage the fact that DRAM retains its state for several seconds after power loss. To exploit this vulnerability, the adversary will power off the device and transfer the remaining data prior to the loss of state. Afterwards, the adversary may tool the BIOS to run an application that transfers all contents of memory across the network to an alternative device, and complete the transfer by powering on the device. If the adversary does not have permission to update BIOS, they may physically detach the memory device and re-attach it to an adversary controlled platform and copy its contents locally.

Write. Much like read-based vulnerabilities, an adversary may use one of the several available channels to inject write-traffic to the memory controller to corrupt data. Even without injecting traffic, however, an adversary with physical access to a memory device may perform targeted corruption techniques on a memory device using radiation [170, 171, 175] or magnets [128].

Even without physical access to the device, an attacker can leverage properties to trigger leakage-induced corruption of memory from software [138, 168, 124, 290, 131]. Accessing a memory location in a tight loop can cause corruption in adjacent words within the memory bank (i.e., the cache line associated with an address). For example, a Rowhammer attack [138] that uses x86 assembly is demonstrated in Listing 2.1. The program, in a tight loop, dirties the data at a given address and asynchronously flushes the provided address from the cache hierarchy in line 4. Note, this program is simple and requires no privileged instructions, so it can be performed by an adversary who can run software on the same device as a victim. This may be because both adversary and victim are running some software on a remote device (i.e., in the cloud) or the adversary is running some guest software on the victim's device (i.e., a webpage).

Listing 2.1: Sample Rowhammer attack.

```
1 char *addr;
2 while ( 1 ) {
3     *addr += 1; // dirty the data at addr
4     asm( "clflush %1" : "r" (addr) ); // x86 instruction to flush addr's cache line
5 }
```

The task of defending against individual tasks is intractable. There are several software gadgets that can be leveraged to perform Rowhammer attacks [191], so detection mechanisms in software are insufficient. Furthermore, new attacks (i.e., RowPress [168] and RAS Clobber [131]) have recently demonstrated comparable capability to Rowhammer, but have different access patterns. Thus, the emergence of new, distinct attacks means defending against these attacks individually in hardware is impractical and insufficient.

2.2.3 Overview of Candidate Threat Models

Data on the processing chip is not subject to the same vulnerabilities as a memory device, so data on-chip (i.e., in the register and cache state) is trusted. That is, the chip boundary also serves as the boundary of the trusted computing base. Data in memory is subject to the vulnerabilities of a memory device. Given the heterogeneity of likely deployments, no assumption about the attacker can be made. That is, an adversary

may have physical access to the device and may run programs with arbitrary permissions on the same platform as some victim. The victim application is assumed to be secure from software-based exploitation at runtime.

Threat Model 1 Given the access of an adversary, a secure memory should be encrypted, but the integrity of data is out of scope. That is, the role of an MEE is merely to ensure that data is safely encrypted prior to its storage in the untrusted memory device, and to decrypt data as it is fetched from the memory device for safe on-chip processing.

Threat Model 2 The privacy and the integrity of data should both be guaranteed by the MEE, but the integrity mechanism does not require authentication. In this threat model, the MEE is responsible for storing encrypted data in memory alongside some MAC associated with the state of the data. Storing data entails both the encryption of data and maintaining the MAC of the data in memory. When that data is later fetched, it is decrypted and its state is authenticated against the state of the MAC. If the authentication is successful, then the decrypted data can be returned across the chip boundary for safe processing. Otherwise, a hardware panic is thrown.

⋆ **Threat Model 3** ⋆ In addition to ensuring the privacy and integrity of data in the untrusted memory device, the MEE is responsible for guaranteeing the integrity of MACs stored in memory. To do so, it must deploy an integrity tree to establish a root of trust on the processing chip. Authenticating data entails both its authentication against the MAC and the authentication of the MAC against its parent nodes in the integrity tree. In this threat model, storing some data in memory entails updating the associated MAC and the integrity tree path that protects the MAC. Ultimately, this dissertation assumes this to be the threat model.

Threat Model 4 Beyond protecting the MACs with an integrity tree, no information should be leaked by the secure memory via side channel. That is, the victim behavior when accessing a secure memory should be oblivious to an adversary with access to the secure memory.

2.2.4 Threat Model Motivation

Consider the following proposition: if data in memory is encrypted, then the precise state of the data cannot be gleaned (i.e., proposed threat model 1). As such, an adversary copying secret data does not reveal the sensitive state, and the malicious ends cannot be achieved. Furthermore, seeing as the sensitive state is now private, any targeted corruption of the data seems to be mute. That is, the attacker cannot determine what to corrupt given the privacy of the data. Thus, they are capable of *blinded random corruption*.

As it turns out, blinded random corruption is a sufficient basis for an attacker to perform significant attack. In particular, it has been demonstrated that performing undetected random corruption of encrypted data can allow an adversary to bypass a login program to grant themselves access to a device on which they otherwise do not have permissions [38]. Furthermore, given the access of the adversary in the threat model, while corruption is likely to be random, the possibility of corruption may not necessarily be blind. This is dangerous, as corrupting data to arbitrary values can have dangerous implications for targets of certain data. For example, corrupting the `NULL` terminator of a string can dynamically trigger a buffer overflow

vulnerability [154]. Blindly corrupting an index can trigger an out-of-bounds access [63] that may either crash a program's execution or trigger invalid dereferences to leak sensitive data.

In addition, an encrypted memory remains susceptible to *replay attack* [81, 247, 93, 248]. Suppose an adversary can read the state of an encrypted memory [102]. Although sensitive data in this memory is private from the adversary, they are able to see the state of both the encrypted data and any metadata used to encrypt that data. If the data is later updated to some new state, the adversary can leverage the corruption vulnerabilities of the memory device to rollback the state of the data to its earlier state. This attack is powerful as the adversary can set some known data to the encrypted memory and record its state. From here, they may replace the data at some sensitive address with their known, dangerous payload.

To account for the potential corruption of data, a MAC associated with the data state may be stored in the memory device (i.e., proposed threat model 2). As such, any corruption of data will be detected by a mismatch of the computed and expected MAC values. However, maintaining a MAC in memory is still susceptible to replay attack. Ultimately, the replay attack vulnerability is a function of having a root of trust outside of the vulnerable device. Seeing as the MAC is subject to corruption in this threat model, it is possible for an attacker to replay the state of both the data and its associated metadata (i.e., the MAC) to the prior state. The MEE has no way to distinguish between a legitimate pair of data and MAC as compared to some replayed data and its associated replayed MAC.

This dissertation assumes that MACs are protected with an integrity tree (i.e., proposed threat model 3). The fourth proposed threat model (i.e., an oblivious secure memory) is deemed too conservative in practice [99] outside of a limited set of lightweight use cases [14]. While some prior work has explored the information leaked via side channel in secure memory hardware [270, 57, 271], mitigating an adversaries ability to learn which locations in memory are used, when they are used, and how often they are used requires extensive and arduous efforts [314, 227, 208, 232]. Furthermore, given that on-chip resources suffer from side channel vulnerabilities [300, 285, 216], an adversary can determine the access pattern of a victim through other means[1]. Thus, the vulnerabilities of a memory device exposed to an adversary warrant efficient protection against corruption-based attacks. As such, while hiding information leakage in a secure memory serves as an ideal case, efficiently defending against leakage of data contents and corruption of its state is a more pertinent issue.

2.2.5 Threat Model Formalized

This work assumes a well understood and studied threat model [298, 225, 151, 317, 301, 86, 224, 284]. Data in memory is untrusted and must be kept private with its integrity guaranteed by some trusted value kept on-chip (i.e., an integrity tree root). The chip boundary serves as the boundary for the trusted computing base. Side channels are out of scope.

2.3 Privacy

In order to guarantee the privacy of data, data must be encrypted prior to its transmission across the trusted boundary. That is, data must not be in plain-text state prior to its storage in a vulnerable, untrusted memory

[1]To account for this, commodity deployments of such defenses that do conform to this threat model limit secure execution to secure co-processors that is free of any shared resources and where any processing components are provably isolated and free from information leakage (e.g., timing, differential power analysis, etc [14].

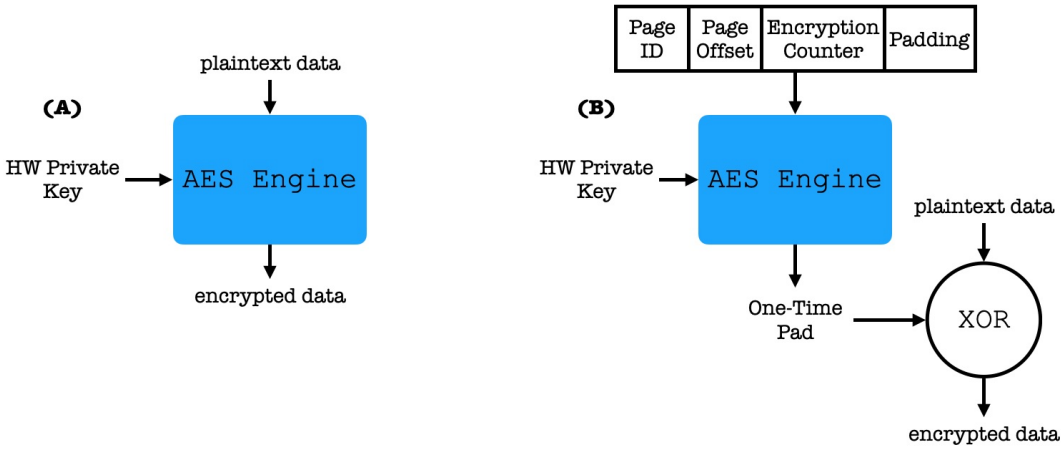


Figure 2.1: Encryption methodologies. Direct encryption (Ⓐ, left) sends data directly to the AES engine to produce encrypted data. Counter-mode encryption (CME, Ⓑ, right) sends an input seed to the AES engine to produce a one-time pad (OTP), which is XOR'ed with the plaintext to produce the encrypted data. In both methodologies, using the ciphertext as input reproduces the plaintext.

device.

2.3.1 Encryption Methodology

Generally, encryption algorithms can be classified as one of "direct" or "counter-mode" encryption algorithms. These approaches are highlighted in Fig. 2.1.

Direct Encryption A direct encryption algorithm, such as the Advanced Encryption Standard (AES) [211], transforms data from its plain-text state to a cipher-text with a hardware private key prior to the storage of the cipher-text in memory [96, 160, 161, 75]. Later fetches for that data uses the same on-chip engine in the MEE to decrypt the cipher-text back to plain-text.

Counter Mode Encryption A counter-mode encryption (CME) instead uses AES to produce a unique *encryption pad* which can be XOR'ed we the plain-text data to produce the associated cipher-text [164]. To produce a secure encryption pad, CME applies the direct encryption methodology (i.e., AES engine in the MEE) to an input *seed* associated with the data as opposed to applying it to the data directly [164]. The AES specification demands that, when encrypting the same data multiple times, a unique seed is used [211] to ensure that an attacker cannot recover the private AES key. Generally, the encryption seed should be both *spatially* and *temporally* unique. That is, each data word should have a unique seed and updates to the data should result in a new seed for that address. Thus, the encryption seed is composed of a unique *counter* that reflects the state of the data comprises part of the input and the address of the data. Whenever data is accessed, the encryption counter is incremented so that the produced pad is unique. That is, the output produced by the AES engine from the encryption seed is a *one-time pad* (OTP).

Generally, there are three encryption counter methodologies that could be used in CME: a single global counter, per-block local counters, or split counters [298]. While using a global counter is appealing (the counter can be maintained on-chip by the MEE and the encryption pad can be precomputed [237]), a secure memory implementation using a single global counter is impractical. The produced encryption pad changes

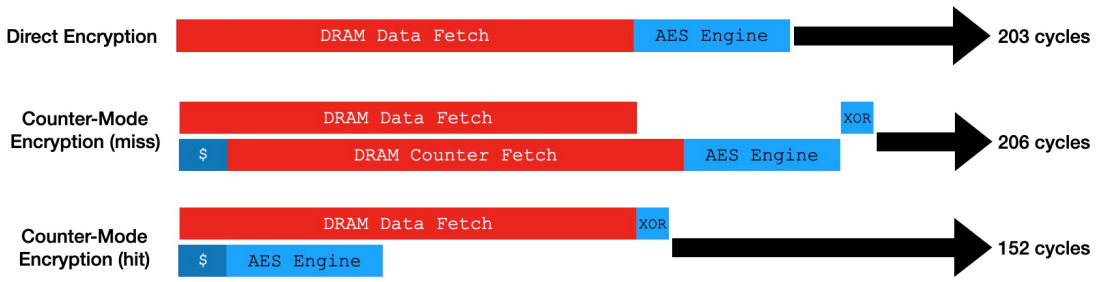


Figure 2.2: Latencies to decrypt fetched data. Assumes 150 cycles to access DRAM, 53 cycles to use the AES engine [147], 2 cycles to access the cache, and one cycle to perform an XOR. When using Counter-Mode Encryption (CME), the counter and data are fetched in parallel.

any time a global counter is incremented, so all data encrypted with that pad must be decrypted prior to its modification, and then re-encrypted with the new pad. Alternatively, a per-cache block local counter can be used to achieve a similar scheme. This approach is practical, as updating a block only entails the re-encryption of that block. That is, because each data word in memory has its own encryption counter, each word will have an associated tag, and updating one word/counter will not impact the produced encryption pad of the other data. Unfortunately, this requires maintaining a local encryption counter for each word; given the limited space on-chip, storing each of these counters in the MEE is infeasible. Consequently, local counters need to be stored in memory as "metadata" for encryption. It has been proven that merely knowing this metadata is insufficient to leak the state of the stored data [30].

In practice, secure memory literature and implementations have converged on "split local counters" [298] as the basis for encrypting data. This implementation elegantly solves two key problems associated with maintaining a local counter associated with each cache block: ① there is spatial overhead attributed to storing each of these counters in memory, and ② a seed may still be duplicated if the counters overflow. Handling an overflow requires regenerating the hardware private key and re-encrypting all data in memory (e.g., equivalent to incrementing a single global counter), so ideally this is an infrequent operation. However, these two features of CME are at odds with one another; shrinking the counter size can reduce spatial overhead but increases the ease of triggering private key regeneration in the MEE. Split counter mode encryption proposes maintaining two local counters per data at different granularities. Each data word (64 bytes) has a 7-bit "minor" counter, and each page (4kB) has a 64-bit "major" counter. As a result, each cache block of encryption counter metadata is associated with a page of application data. In the split counter scheme, the (7-bit) minor counter is incremented by default. On overflow, the major counter is incremented. Seeing as the major counter is associated with all data in the page, each of these data must also be re-encrypted. Upon doing so, the other minor counters in the page may also be set to zero to reduce the likelihood of incrementing the major counter multiple times.

Formally, split counters require 8-bits of spatial overhead per data word (64 bytes). Furthermore, while incrementing a major counter requires re-encrypting all data in a page, a major counter overflow requires at least $2^7\times$ more accesses than a single 64-bit local counter (the major counter is only incremented on minor counter overflow). Thus, split counters alleviate both sides of the limitation trade-off space imposed by per-block local counters.

Implementing encryption algorithms in the MEE comes at a performance cost by putting decryption, an

expensive operation, on the critical path of a cache block fetch [77, 164]. To account for this, encryption counters in CME may be cached in the *metadata cache* in the MEE. Furthermore, the OTP can be computed in parallel with the data fetch so long as the MEE has the encryption counters. Performing the OTP computation in parallel with the data fetch can have huge benefits in terms reducing the latency on the critical path of fetching data as compared to direct encryption [236, 237, 299, 248]. Fig. 2.2 shows the latencies associated with fetching data in a direct-encryption memory as compared to using CME on a metadata cache hit and miss.

2.3.2 Encryption Performance

This section empirically explores the implications of various encryption design decisions on end-to-end runtime overhead. In particular, it explores three schemes: ① no security (which serves as a baseline), ② direct encryption , and ③ counter-mode encryption (CME). Direct encryption and CME are described in Sec. 2.3.1. Within these configurations, this analysis explores the impact of several design decisions.

This analysis uses the GAP [26] benchmark suite with synthetic graphs of various input sizes[2]. In particular, the suite consists of five graph analysis benchmarks: `bc` computes the "betweenness centrality" of two graphs; `cc` computes the "connected components" of the graph; `tc` performs "triangle counting" in the graph; `pr` performs a "page rank" computation on the graph; and `bfs` performs a "breadth-first search" computation on the graph. This analysis is done with varying input sizes to demonstrate the how the approaches scale as the workloads increase in memory intensity.

Encryption Overhead Fig. 2.3 shows the overhead of using a default configuration of direct encryption and counter-mode encryption (CME) relative to a memory controller without secure memory. In particular, direct encryption exhibits up to 80% overhead (38% overhead on average) relative to having no security. A memory controller that implements CME with a 64kB metadata cache exhibits up to 15% overhead (4% overhead on average) relative to having no security. The analysis shows that there is a direct relationship between the memory intensity of a workload and the overhead of the encryption. That is, as the graph input size increases, so does the overhead of performing encryption as they tend to have larger memory footprints. If an application exhibits a larger memory footprint, then accesses memory at a higher rate as fewer accesses will hit in the

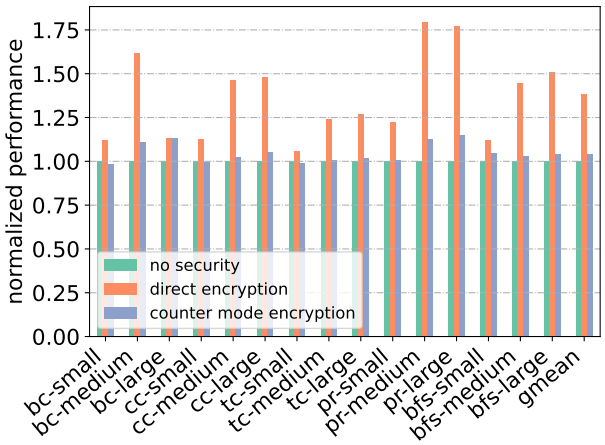


Figure 2.3: Performance overhead of a secure memory with direct encryption and counter-mode encryption relative to an insecure memory controller.

[2]This is the default configuration to do this performance in gem5. Note, in this evaluation "small" refers to "test" in gem5, "medium" refers to "small", and "large" refers to "medium." The resource for the GAP benchmark suite does not contain a default "large" input size. The test input graph has 2^{13} nodes, the small input graph has 2^{21} nodes, and the medium input graph has 2^{23} nodes.

on-chip cache hierarchy. Memory requests in an encrypted memory have additional work on the critical path of the memory fetch, so more memory requests (i.e., misses in the cache hierarchy) will result in longer performance. This is formalized in the observation below:

Observation 1: The overhead of memory encryption is a function of the rate at which the application accesses memory.

Note, the results indicate that direct encryption incurs larger overhead than CME. This is a function of the fact that, on read requests, the OTP can be produced in parallel with the fetch for data (described in Sec. 2.3.1). Furthermore, seeing as the encryption counter can be fetched from the on-chip metadata cache, the latency of this operation is hidden on hit cases. The workloads in which the overhead of CME is most evident are those with larger footprints, in which the efficacy of the cache is reduced. Fig. 2.4 shows the hit rate for encryption counter fetches in the metadata cache across the workloads. Note that there is a strong negative correlation between CME overhead and metadata cache hit rate. Consider the `pr` benchmark. When using the small input size, the performance overhead of CME is less than 1%, and the metadata cache hit rate is over 99%. When increasing the graph size to the medium input, the metadata cache hit rate decreases to 60% and the overhead of CME becomes 12% relative to the insecure baseline. With the large input graph, the metadata cache hit rate for encryption counters decreases to 30% and the overhead of CME increases to 14.5%. Given this analysis, the following can be stated:

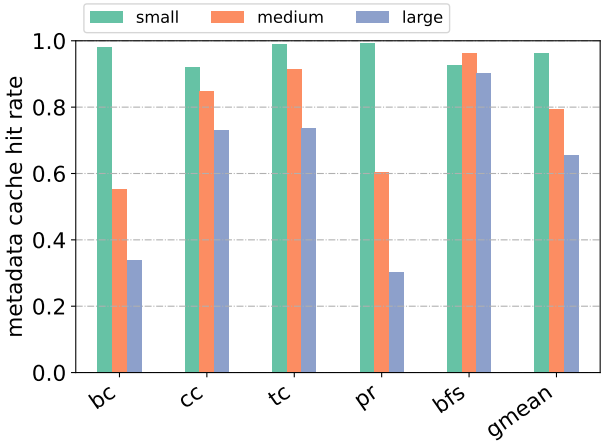


Figure 2.4: Hit rates in the metadata cache for encryption counters across workloads.

Observation 2: CME has per word metadata, so the relative overhead of that methodology is inversely proportional to the metadata cache hit rate.

For the most part, direct encryption serves as an "upper bound" on CME performance. All accesses will need to go through the cipher engine and will have additional critical path latency as a result. However, this is not a tight upper bound. The additional fetch in CME for the encryption counter puts an additional demand on the available bandwidth of the memory device. In cases where the device bandwidth is saturated, the request for data and its associated encryption counter will not be performed in parallel, and the overhead of CME may be larger than in direct encryption. In this evaluation, the additional bandwidth requirement of CME may be was still below the available device bandwidth, therefore the parallelism did not result in a performance bottleneck. Typical DDR4 devices typically support peak bandwidth of approximately 25 GiB/s [1], so the device bandwidth is not saturated in these cases.

Fig. 2.5 shows the average bandwidth utilized by each benchmark and configuration. It shows that

bandwidth requirements increase as workload size increases and that CME typically occupies more bandwidth than the baseline approach. In particular, the memory bandwidth required in a CME memory system requires 2.4× more bandwidth than the no security baseline. This trend is relatively consistent across small, medium, and large input sizes, where the bandwidth overhead of CME is 4%, 22%, and 36% respectively. The bandwidth overhead can be attributed to the additional requests to the memory device for encryption counters. Many of these requests can be serviced by the metadata cache and, in turn, avoid the memory device, but misses and writebacks from the metadata cache must still be handled by the memory device. Thus, the following claim can be made empirically:

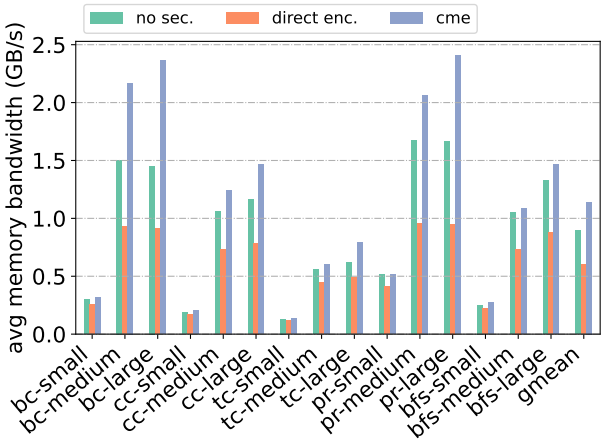


Figure 2.5: Average memory bandwidth utilized by different security configurations in terms of GB/s.

Observation 3: CME requires more bandwidth than a baseline application due to the fetches for encryption counter metadata.

Note, direct encryption actually requires *less* bandwidth than the baseline as the additional time required to cipher certain requests allows the elimination of parallel requests for data at similar addresses. This optimization is described in detail in Appendix A. At a high level, the increased latency will cause more parallel memory requests for the same address and these requests can both be processed by a single memory packet. However, the benefit of this optimization is likely inflated in this study due to a relatively small cache hierarchy (256kB LLC), so there will be more parallelism for equivalent addresses in the memory device. The CME implementation also takes advantage of this optimization (including fetches for encryption counters), but it is not as effective for two reasons: ① ciphers can be performed with lower end-to-end latency, and ② encryption counters still require an additional fetch.

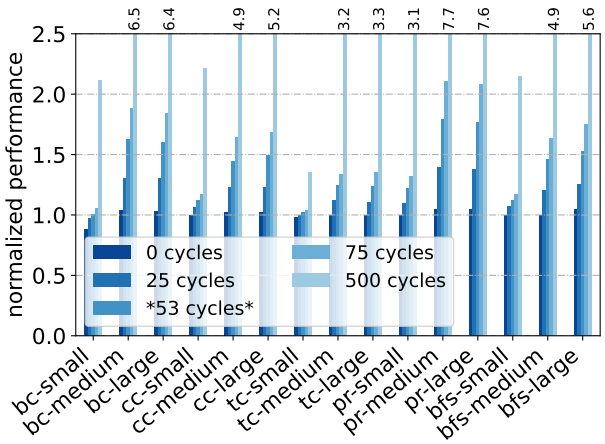
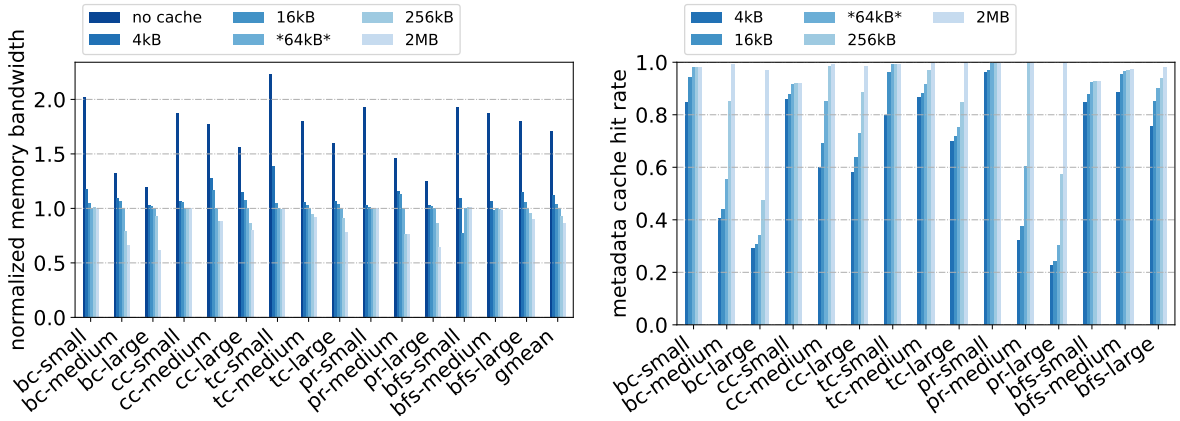


Figure 2.6: Performance overhead of direct encryption with various cipher latencies relative to an insecure memory controller. Note, 53 cycles is considered default.



(a) Bandwidth of CME normalized to default metadata cache size (64kB) with various metadata cache sizes.

(b) Hit rates with various sized metadata caches using CME. Note, a 64kB metadata cache is the default size.

Figure 2.7: Sensitivity study of bandwidth and hit rate relative to metadata cache size.

Cipher Latency Sensitivity To explore the potential performance benefits and degradation of the cipher latency, this section explores the impact of cipher latency on direct encryption performance. This is depicted in Fig. 2.6. The presented results are normalized to the number of cycles to execute the benchmark with no security (not depicted). In particular, the figure shows the number of cycles to execute each workload normalized to a memory controller without secure memory. When cipher latency is eliminated, the overhead of performing direct encryption is relatively negligible compared to no security. On the other hand, the overhead may be as large as 7.7× when cipher latency is dramatically increased. Notably, the cases with the most significant overhead are the those workloads with significant memory footprints. This is consistent with the finding of Observation 1. That is, more latency on the critical path of a memory fetch is more significant for workloads that access memory more frequently (i.e., have a larger memory footprint).

Counter Cache Size Sensitivity This section also explores the impact of metadata cache size on encryption counter hit rates. The aim of this study is to explore the relationship between metadata cache utilization and the behavior of the secure memory system as increasing the size of a cache is a natural way to improve utilization. Thus, metadata cache size serves as a proxy for "utilization," which is explored directly by studying the cache hit rates in Fig. 2.7b.

For the most part, varying metadata cache size does not have a significant impact on end-to-end performance aside from workloads in which the overhead of CME was highest (e.g., an 8% performance improvement when using a 2MB metadata cache as compared to a 64kB metadata cache in the `pr-large` workload). That is, performance improvements are only evident when the impact on metadata cache hit rate is dramatically positive. Fig. 2.7b shows the hit rate in the metadata cache for various configurations with different metadata cache sizes. Consistent with Observation 1, workloads with larger memory footprints tend to have lower metadata cache hit rates, and hit rates can be improved as the cache size increases. For each workload, the hit rate in with a 2MB metadata cache is over 90%.

Effective metadata cache utilization is closely related to the bandwidth utilized by an application. Fig. 2.7a demonstrates the impact of metadata cache size on average memory bandwidth demanded. The

reported metrics are normalized to the default metadata cache size (64kB). As evidenced by the evaluation, not using a metadata cache would result in 70% additional bandwidth demanded on average across the suite. With a 2MB metadata cache, on the other hand, the demanded bandwidth can be reduced by 13%. For workloads in which the bandwidth demands of CME are highest relative to the non-secure alternative (e.g., `large` input sizes), the reduction is by over 25%. Therefore, improved metadata cache utilization most benefits the memory bandwidth for applications with larger memory footprints. Given this analysis, the following observation can be formalized:

Observation 4: As metadata cache utilization improves, the memory bandwidth utilized by secure memory decreases.

Counter Arity Sensitivity Observation 4 implies that improving metadata cache utilization is of paramount importance. Such is the motivation for having the split-counter design in CME. This section studies the impact of various "arities" (i.e., the number of data words that leverage the same encryption counter block) on metadata cache utilization. By default, there are 64 data words per encryption counter block (i.e., 64 seven-bit minor counters). By increasing the arity, the effective "utility" of a metadata cache block is increased as more data depend on it. Fig. 2.8 shows the impact of various encryption counter arities on hit rates in a 64kB metadata cache (the standard size for secure memory). Note that, with unary or binary encryption counters, metadata cache hit rates decrease dramatically. These configurations can be characterized by two features: ① there are only one or two data blocks respectively that depend on each encryption counter, and ② the number of encryption counters needed to protect the memory state must increase. Conversely, 4096-ary encryption counters result in near perfect hit rates in the metadata cache. In these instances, 256kB of data are protected by a single encryption counter. Consequently, it is highly likely that the relevant metadata for an encryption/decryption will reside in the metadata cache in these cases. More formally:

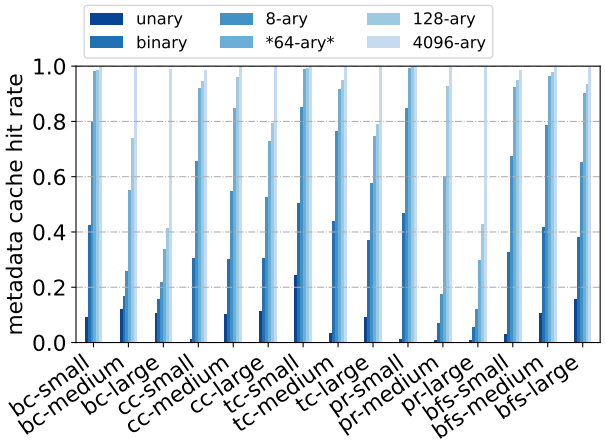


Figure 2.8: Hit rate in a 64kB metadata cache with different encryption counter arity configurations.

Observation 5: Metadata cache utilization can be improved as the utility of metadata cache blocks increases.

2.3.3 State of the Art

The performance overhead of encryption is largely a function of the cipher engine latency. Seeing as the cipher is on the critical path of a memory fetch, a faster decryption implies a lower latency to return data to the processor. In addition, Sec. 2.3.2 highlights how CME can significantly outperform direct encryption,

especially given an effective metadata cache. As a result, the state-of-the-art performance optimizations for encryption typically concern improving the latency of the cipher engine and the utilization of the metadata cache for encryption counter metadata.

For the most part, secure memory assumes an encryption engine utilizing some form of Galois Counter Mode Encryption (G-CME) [177]. To further reduce encryption latencies, prior work, such as ELM [121], lean on improving algorithmically implemented cryptographic primitives. This particular approach is based on *offset-codebook mode* [220] encryption to reduce the number of tweakable block cipher [165] iterations. However, efforts to improve cipher latency are orthogonal to secure memory and secure memory can treat the encryption methodology as a black-box.

Alternatively, seeing as CME performance is closely tied to metadata cache utilization, several works have considered strategies to improve the utility of encryption counter cache blocks. As described, split-counter mode encryption [298] proposes a scheme in which every 64-byte data word is effectively protected by 8-bits of encryption counters (i.e., a private 7-bit minor counter and a shared 64-bit major counter with 63 other words). To improve the utility of an encryption counter cache block, more data would need to be protected by the same amount of metadata (i.e., the metadata size per 64-bytes of data must be reduced from 8-bits).

To achieve this, several prior works have made observations about the contents of encryption counters at runtime. For instance, *morphable counters* [224] observes that encryption counters largely have the same contents due to infrequent access of their associated data. As such, seeing as many counters are either "0" or "1", the size allocated to these counters can be compressed. This scheme proposes two potential states for encryption counters (i.e., compressed or decompressed) denoted by a reserved bit from the major counter. If the minor counters are infrequently used, then the space allotted to them can be decreased and more counters can be stored within the block. Similarly, *common counters* [192] makes the observation that GPU memory is often only ever written to once when the data is set on initialization. As a result, this work proposes even further compressed encryption counters for data that is explicitly marked as read-only by the application and/or GPU. This work has led to several follow up works on building secure memory for these workloads [307, 2, 287, 193, 137].

2.4 Hashing

The MEE guarantees the integrity of data (i.e., that data hasn't been corrupted) in memory by maintaining *metadata* associated with the state of the data [34]. Data privacy alone is insufficient; an attacker can corrupt some cipher-text such that the MEE produces garbage bits post decryption rather than the intended data [38]. For this reason, the MEE also maintains a set of cryptographic hashes associated with the data to ensure that data has not been corrupted during its storage.

Determining the integrity of some value is a well studied problem. A message authentication code (MAC) gives a mechanism for some party to validate the state of some untrusted value [160, 161, 29]. Using a MAC serves as a "cryptographic checksum" [29] for some data that may have been corrupted in the presence of an untrusted environment, which makes it a natural candidate for secure memory. As such, the integrity of data can be checked despite its storage in an untrusted memory device so long as both the data *and* its associated MAC are stored.

A secure MAC is one in which an attacker cannot produce a legitimate code for some arbitrary data. As such, deriving a MAC is functionally equivalent to using a *cryptographic hash function* [241]. That is, the

hash function should have the following properties: ① the MAC is computed from a one-way hash function (i.e., if the hash of x is $H(x)$, then it should be computationally infeasible to find x from $H(x)$ [180]), and ② the MAC should be produced by a collision resistant hashing function (i.e., if the hash of x is $H(x)$, it should be computationally infeasible to find x' that also produces $H(x)$ [180, 179]). In particular, secure memory literature and deployments have converged on a keyed hash-based message authentication code (HMAC) [29, 267, 140] to facilitate the checking of data integrity. HMACs have been proposed as practical relative to other protocols such as NMACs [29] by considering the hashing algorithm as a black-box. Further innovation of MACs (e.g., CMACs and KMACs [306]) have been proposed with enhanced security guarantees, but come with performance limitations.

Innovation in MAC security and efficiency is important, but developing these protocols alone is orthogonal to the secure memory problem. Instead, the primary concern in secure memory is the *utilization* of the MAC values. Consider the MAC based secure memory protocol proposed in Gilmont, et al [96]. To authenticate some data entails: ① fetching that data and its associated MAC, ② computing the MAC from the fetched data, and ③ comparing the computed MAC to the stored (expected) MAC. So long as the computed and expected MAC values are consistent, the MEE assumes that the data hasn't been corrupted. Such an implementation is sufficient to defend against *spoofing attacks* [81] (i.e., replacing legitimate data with arbitrary malicious data).

2.5 Integrity Trees

Unfortunately, using a MAC alone is insufficient to make this guarantee as MACs are subject to replay attack [248, 161, 222, 298]. Given the threat model, attackers have access to the state of data and metadata in memory. Suppose that at t_0 an adversary reads data D with MAC M_D. At t_1, the data and its associated metadata are legitimately updated to D' and $M_{D'}$. Given that the adversary has the ability to corrupt the data, they are able to revert that data from D' back to D and H_D. This is a powerful attack capability, as adversaries can use this vulnerability to leak and/or perform targeted corruptions of data [186].

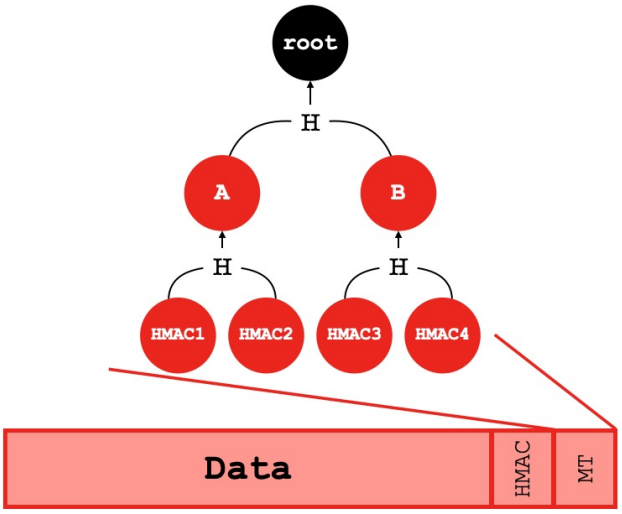


Figure 2.9: A Merkle tree built over untrusted memory. HMACs of the data serve as the leaves of the tree, and multiple HMACs are hashed to produce the parent node. The root of the tree is a single value that can be stored on-chip.

Ultimately, this vulnerability is born out of the fact that the MACs, much like encryption metadata, need to be stored in the untrusted memory device. To account for this, MACs themselves require some authentication. Fig. 2.9 shows how this can be achieved. Seeing as MACs are untrusted, they require authentication. This can be achieved by maintaining the MAC of a set of untrusted MACs, and can repeated recursively. Notice that, in such a scheme, the MACs are arranged hierarchically such that they form a *Merkle tree* [178].

To this end, a Merkle tree serves as a strong basis for a secure memory [34, 93, 298, 222, 247]. Merkle trees have been applied in a variety of contexts in order to trust data that is otherwise subject to potential tampering [172, 46, 25, 314, 294, 265]. In particular, the construction of a Merkle tree implies that only trusting the tree root is sufficient to authenticate the integrity of all untrusted data in the underlying memory device [160, 99, 75, 99]. As such, storing this single root of trust in trusted hardware (i.e., in the trusted computing base on-chip) means that any authentication against it can in turn be trusted. To authenticate some data against a Merkle tree: ① the MAC is computed from the data, ② the MAC is authenticated against the stored MAC, ③ the word containing the MAC becomes the data to authenticate, and ④ the process is repeated until an authentication can be made against the Merkle tree root. Note, the Merkle tree root reflects the state of the entire underlying memory.

Storing secure memory metadata in an on-chip cache can significantly benefit secure memory [298, 222, 247, 256, 93, 81, 225, 257, 224, 214, 276, 274, 275, 151, 150, 252, 23, 24, 8, 301, 88, 89, 86, 87, 107, 108]. The benefit of doing so is two-fold: ① as described in Sec. 2.3, metadata can be fetched at lower latency than going to memory, and ② the metadata cache resides on-chip, so cached values are trusted [94]. That is, when authenticating some data, integrity tree nodes need to be fetched from the leaves up to the first metadata cache hit (or the root of trust if none of the path is in the cache) due to an expansion of the trusted computing base (TCB). This benefits *latency* as fetching metadata on the critical path of a memory operation can be achieved in fewer cycles if that metadata resides in the cache. Furthermore, the metadata cache benefits *bandwidth* by reducing the number of fetches for security metadata that need to go to the device, thereby protecting memory device availability.

Given that the metadata cache benefits both of the key performance limitations in secure memory, the study of its efficacy is paramount. Early secure memory proposals considered storing encryption and/or integrity metadata in the on-chip last level cache (LLC) [94, 93, 247, 298], but metadata pollutes these caches and reduces LLC hits for data. This implies more misses for data in the LLC, and more requests for data were sent to the memory device as a result. Such is the motivation for having a separate cache dedicated particularly for secure memory metadata [222]. In fact, several works have proposed a dedicated cache per metadata type (i.e., integrity tree nodes, encryption counters, MACs) to capture improved cache utilization without having cross-type interference. MEEs that have been deployed in practice, however, tend to have a single cache dedicated for all metadata types [99, 109].

Metadata accesses do not conform to the same heuristics as application data [150]; accessing an uncached integrity tree path is a pointer-chasing operation from leaf-to-root. As such, data accesses to random, uncached paths in the integrity tree are unlikely to exhibit locality until close to the root. On the other hand, a data access that does exhibit locality may share an ancestor in the integrity tree closer to the leaves. The tension between these two cases implies that several cache blocks have limited utility after their initial fetch, but knowing what these cache blocks are is non-trivial. Furthermore, to ensure that the tree root is consistent with the state of the data, updates to the data state entail updating the entire leaf-to-root path. To load each of these blocks into the metadata cache may cause further interference in what is currently cached.

Several works have considered placement and replacement for metadata caches. Given this access pattern, traditional caching do not extend to metadata caching. For example, Belady's optimal cache replacement [28] does not apply to metadata caches [150]. The intuition for this phenomenon is that each potential replacement ultimately changes the access pattern for future authentications. As a result, some works have considered

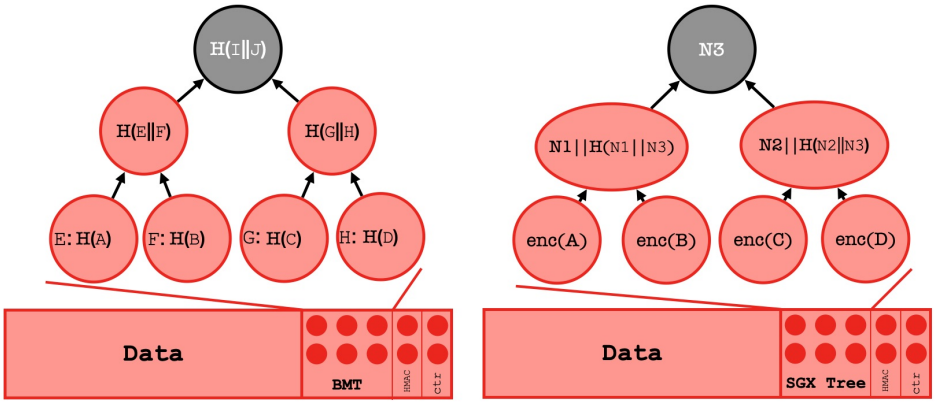


Figure 2.10: Bonsai Merkle Tree (BMT, left) and a tree of counters (SGX style tree, right). Parent nodes in a BMT are the concatenated hashes of their children. Leaves in a tree of counters are encrypted with the nonce in the parent node. Inner tree of counter nodes maintain a MAC of its state using the nonce from its parent node. In both, the leaves protect encryption counters rather than data.

alternative approaches to choose targets for metadata placement and replacement [149]. For example, the set of subtree roots in the BMT in the metadata cache may be updated based on child access behaviors [86]. Doing so efficiently, however, is difficult to achieve outside of extreme cases (i.e., a write-through cache for non-volatile memory). Alternatively, lazily propagating updates through cached nodes can help avoid the costly operation of updating the tree root [248]. To achieve this, metadata cache writebacks are coupled with loading the parent of an evicted value into the cache with the updated parent node. In doing so, updates to the data state are always reflected in the TCB even if the BMT root is not up-to-date. Unfortunately, such a scheme can create a cascading effect; loading the parent of an eviction and/or writeback target may trigger another eviction.

2.5.1 Data Structures for Integrity Trees

Standard Merkle Tree One possible counter integrity scheme maintains a subtree of the Merkle tree for secure memory that protects the entire memory state. This includes both the state of memory and of encryption counters [93]. In such a scheme, fetching data from the secure memory entails authenticating both the encryption counter and data against the Merkle tree. Once the integrity of the encryption counter is verified, the MEE can decrypt the data prior to returning the plain-text to the processor. This procedure guarantees the integrity of the data and encryption counters by directly authenticating them against the root of trust (i.e., the Merkle tree root) stored in trusted hardware (an on-chip register) in the MEE.

Bonsai Merkle Tree Using a standard Merkle tree means that each fetch for application data requires two authentications (the data state and the encryption counter state). This entails fetching two paths through the integrity tree, which imposes a latency penalty and constrains the bandwidth available in a memory device. Such is the motivation for a Bonsai Merkle Tree (BMT) [222].

In a BMT, encryption counters act as the basis of the integrity tree, and encrypted data has its integrity protected by MACs that are distinct from the integrity tree (which still use encryption counters in the computation). Inner nodes in the BMT are the concatenated 8-byte HMACs of their associated children. The major-minor counters from split-counter CME are the leaves, each of which is 64-bytes. The HMAC

of these values are computed and stored as the parent node in the BMT. Seeing as the produced HMAC is 8-bytes, the concatenated HMACs of eight major-minor counters comprise the parent node. That is, the BMT is 8-ary. To perform an authentication using a BMT, the encryption counter is fetched and has its integrity authenticated by the Merkle tree. In parallel, the encrypted data has its integrity authenticated by its associated MAC. If the MAC authenticates the data, then the data's integrity can be indirectly guaranteed so long as the encryption counter hasn't been corrupted.

The insight of this approach is that the integrity of the plain-text data is *indirectly* protected by the BMT. If the plain-text data produced from the encryption counter is authenticated by the MAC, then it must be the case that neither the data nor the encryption counter was tampered. As such, authenticating data requires fetching the (untrusted) encryption counter to decrypt the data; if the Merkle tree state authenticates the plain-text, then it is safe to respond to the processor. Furthermore, this scheme couples the role of the encryption counters with the integrity authentication protocol. Note, MAC collisions for the same data can occur in this scheme if the encryption counters overflow as the basis for the MAC computation is consistent in both states, but this only happens after 2^{71} accesses by using split-counter mode encryption.

Trees of Counters Authenticating encryption counters in a BMT requires extensive HMAC computation. Each level of the integrity tree is composed of HMACs of its children, so the authentication of a node at any level requires full MAC computation. These computations can be expensive, and performing multiple authentications in parallel requires complex logic [59, 58][3].

On the other hand, a parallel authentication tree (PAT) [103] is a tree of counters (nonces) and hashes rather than just hashes. In particular, a PAT node consists of a MAC and a series of counters (nonces) associated with its children. The MAC value in a node is computed from the associated nonce value. Much like a BMT, a PAT uses encryption counters as its basis. Thus, when an encryption counter is updated, the associated nonce is updated in the parent node. This construction of nodes creates a dependence in MAC computation between a child node and its parent, which allows for authentication to occur against the root of the tree and the global nonce. Similarly, a tamper-evident counter tree (TEC-Tree) [81] describes a tree of counters in which tree nodes are encrypted with the trusted root nonce. As such, authenticating data with such a scheme entails decrypting the node prior to comparing the tag against the state of the payload. The data structure for a tree of counters is portrayed in Fig. 2.10.

The motivation for the PAT organization is to simplify the procedure of parallel updates to the integrity tree. In order to parallelize updates in the BMT, complex logic manages on-chip logs of pending authentications in the MEE before and after the HMAC of the node is computed. On the other hand, parallel updates in the PAT are trivial: two nonces and the associated MAC in a node can be updated and computed concurrently. So long as the updates are visible to all updating parties, any computation of the MAC value will be correct.

[3]The intuition for how to perform parallel authentication with a tree of hashes (i.e., standard Merkle tree or BMT) is similar to that of a miss status holding register (MSHR) in a cache [141, 266]. Pending authentications and updates to the integrity tree state need to be maintained in an on-chip log. Whenever a new authentication or update is requested, this on-chip state must be checked first such that the two requests can be handled by the same fetch and hash computation. If two updates to the same node are detected in parallel, the second one completes the update for the rest of the tree branch.

2.5.2 Tree Updates with Metadata Caches

To authenticate the state of data, the root of trust must be kept consistent with the state of the data. Any update to the data state must similarly update the metadata state and the root of trust. However, doing so can result in a high volume of metadata accesses. As such, prior literature has proposed both eager and lazy schemes to update the trusted state. In a BMT, updates to the state of the data entail updating the encryption counter. Consequently, the hashes of the encryption counters (and the other hashes in the leaf-to-root path through the BMT) will change. In a tree of counters, the nonces in inner nodes associated with the updated child must be incremented. This requires recomputing the MAC of each of the nodes whose nonces are updated.

Eager Update One possible scheme would be to eagerly update the root of the integrity tree any time the data state is updated. All nodes in the leaf-to-root path would be updated on every update to the data state. Seeing as the root is updated, this scheme ensures that the trusted data and metadata state is always entirely consistent with the untrusted state.

This requires updating the leaf-to-root path *atomically*. That is, all updates to the path must appear at the same time to ensure that data races for the metadata state do not occur. Typically, this is implemented in the memory controller by treating the write-pending queue (WPQ) as a log in which modifications are flushed from the MEE atomically. Changes to the metadata state are placed in the WPQ upon update, but they are not flushed to the memory/cache state until a "ready bit" is set. This allows all metadata values (i.e., new hashes in a BMT or nonce/MAC values in a counter tree) to be computed by the MEE with the requisite engines without worrying about the atomicity of the computations themselves.

Lazy Update Eagerly updating the integrity tree root demands a significant number of requests to the memory system. Each node update requires a write to the system for the whole leaf-to-root path, and this can put a significant strain on the available device bandwidth. To account for this, a MEE may implement a lazy integrity tree update scheme.

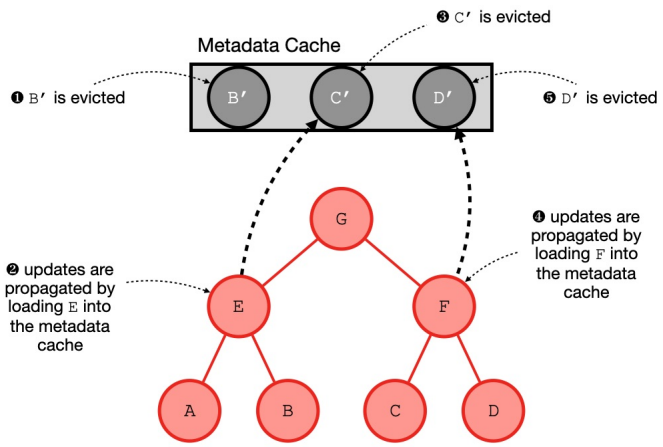


Figure 2.11: Propagation of lazy updates in the metadata cache that triggers cascading update effect. Note, the original update will still need to update G after each of these changes as a trusted value still hasn't been updated.

Instead of updating the tree root on every data update, a lazy update scheme only updates the path from the leaf towards the root up until the first metadata cache hit. In doing so, the first accessed trusted state for authentication is kept consistent with the state of the untrusted state of the data. If some dirtied metadata is later removed from the trusted state (i.e., evicted from the metadata cache), then the updates to that state must be propagated

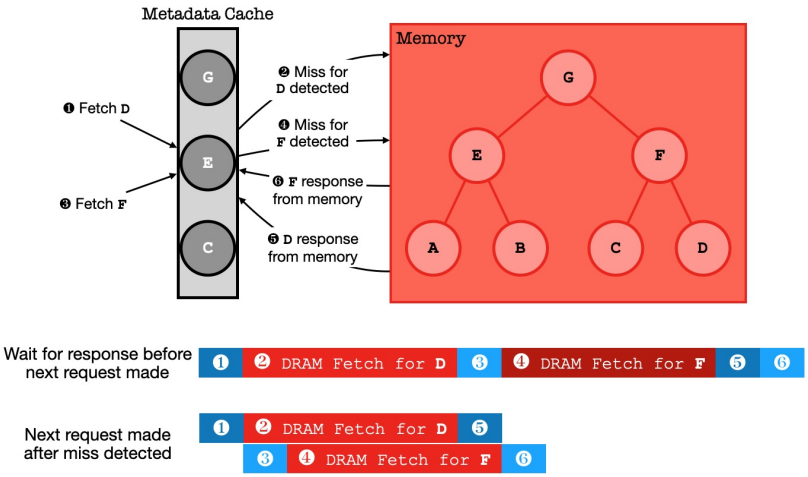


Figure 2.12: Benefit of fetching integrity tree nodes upon detected metadata cache miss.

upwards in parallel with the eviction so that the trusted state is still kept consistent with the state of the data.

Note, this approach to updating the integrity tree comes with two key limitations: ① knowing which nodes are in the metadata cache requires additional MEE complexity to track to metadata state prior to update, and ② propagating changes upwards can result in cascading evictions from the metadata cache. To expand on ②, Fig. 2.11 shows how such a cascading effect can occur. When a node is evicted from the cache and its parent is not in the cache, the parent node must be fetched and placed into the cache. In doing so, another node may be evicted from the cache and the process repeats. In the worst case, lazily propagating changes upwards can evict and replace the entire metadata cache state.

There are two potential strategies to propagate these changes upwards towards the integrity tree root. One option is to propagate these changes up to the next trusted value (i.e., a parent node in the metadata cache). Alternatively, propagated changes may be eagerly applied up to the root. Given the implementation difficulties of lazy update propagation, any eagerly updates to the integrity tree state simplify the MEE logic.

2.5.3 Integrity Tree Traversal

To authenticate data in an untrusted memory, the metadata associated with the data is fetched and used by the MEE to verify its state. The MEE will fetch nodes from the leaves towards the root up to the first trusted value. This is denoted by the first hit in the metadata cache or by the integrity tree root. This procedure is depicted in Fig. 2.12, which also highlights the potential performance benefit relative to waiting on the full response from the initial request. Upon detecting a miss in the metadata cache, the request for the next node in the integrity tree is created. Note, this is possible because the integrity tree is *not* pointer-chasing by default. The parent node of any child can be found by a computation on the address of the data.

This traversal strategy is inherently lazy in nature [79]. With that said, there is a key distinction in the complexity of lazily traversing the tree as compared to updating the tree. Updating the tree requires knowledge of what is in the cache state *before* the request for the node is made whereas fetching integrity tree nodes lazily only occurs after detecting a miss in the cache state. While small, this distinction makes

the resulting logic much simpler.

2.5.4 Integrity Tree Performance

Integrity Overhead To examine the overhead of integrity protocols, this thesis performs a similar study to Sec. 2.3.2 concerning MAC and integrity tree structures. Fig. 2.13 shows the overhead of using a MAC and BMT relative to the CME and no security results from Sec. 2.3. Both of these schemes extend the protections from CME, so the overhead of each approach can be visualized as the difference between this baseline protection.

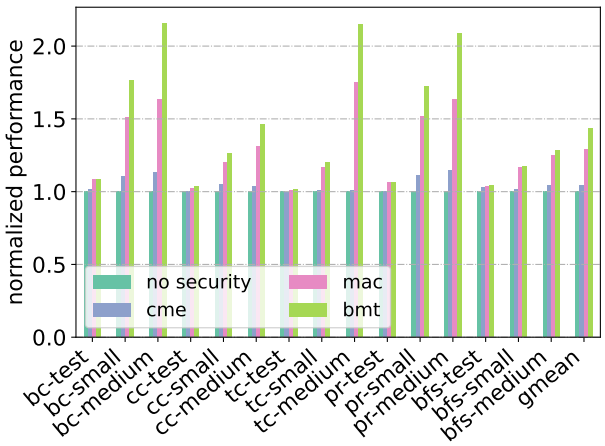


Figure 2.13: Normalized cycles of MAC-based and BMT-based integrity schemes relative to counter-mode encryption (CME) and no security to execute GAP benchmark suite.

Much like the analysis from Sec. 2.3.2, the most significant overhead in the integrity protocols comes from more memory intensive cases. That is, benchmarks like `bc`, `tc`, and `pr` have more memory operations, so the integrity protocol gets exercised on a greater percentage of instructions and requests. In these cases, the overhead of using a MAC is 29% relative to no security and 25% relative CME on average, and 1.75× in the worst case (`tc` large). Using a BMT for integrity provides stronger protection, but comes with 43% overhead relative to the no security baseline and 39% overhead relative to CME on average. In the worst case, a BMT incurs 2.16× overhead relative to the non-secure baseline (`bc` large) and 2.15× relative to CME (`tc` medium).

As shown in Sec. 2.3, performance overhead appears in terms of both critical path latency added to memory fetches and the additional bandwidth required to fetch secure memory metadata. Fig. 2.14 shows the bandwidth overhead of both the MAC and BMT integrity schemes. In particular, the analysis shows that using a BMT results in 79% more bandwidth relative to a non-secure memory system for the GAP benchmark suite on average. This is an additional 53% on top of CME, and is a function of the additional metadata required for the integrity structures. Seeing as the BMT requires more metadata for authentication than a MAC based integrity scheme, it consumes more memory bandwidth.

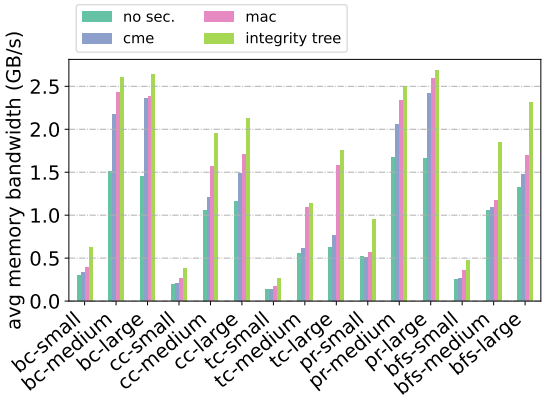


Figure 2.14: Bandwidth consumed by various integrity schemes in the GAP benchmark suite in terms of GB/s.

Bonsai Merkle Tree Performance As demonstrated by Sec. 2.3.2, metadata cache utilization is closely tied to performance. The BMT was originally proposed as a

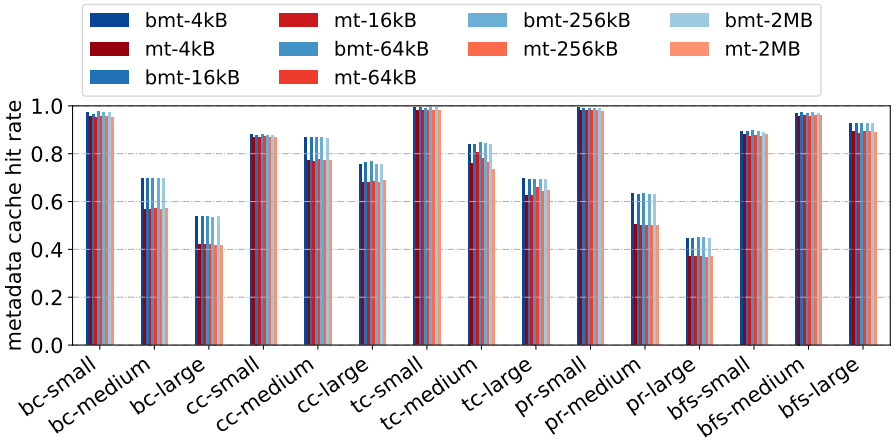


Figure 2.15: Comparison of BMT and MT metadata cache hit rates for various metadata cache sizes.

means towards achieving a denser integrity tree than a standard Merkle Tree over MACs [222]. As a result, caching a BMT is a much easier feat in the equivalent cache space. Fig. 2.15 shows metadata cache hit rates across the GAP benchmark suite for the BMT and Merkle Tree in small to large metadata cache sizes. Although hit rate is largely dictated by the application behavior, the key takeaway from this evaluation is that, on average, using a BMT improves metadata cache hit rate by 5% relative to the alternative.

Tree Arity Study To further express the point of metadata cache utilization, this analysis includes a study of the effect of tree arity on performance. In particular, Fig. 2.16 shows the impact of various arities on integrity tree performance for the GAP benchmark suite. Performance is normalized to an 8-ary BMT, which is default for secure memory. Note, tree arity is a function of the number of hashes that can fit in a single 64-byte data word. Prior work in hash constructions, described in Sec. 2.4, converges on the fact that an 8-byte hash value is sufficiently small and resistant to replay attacks. This analysis assumes that this cryptographic is not a restriction, and studies the performance impact of higher arities (i.e., smaller hashes) if they were sufficiently secure.

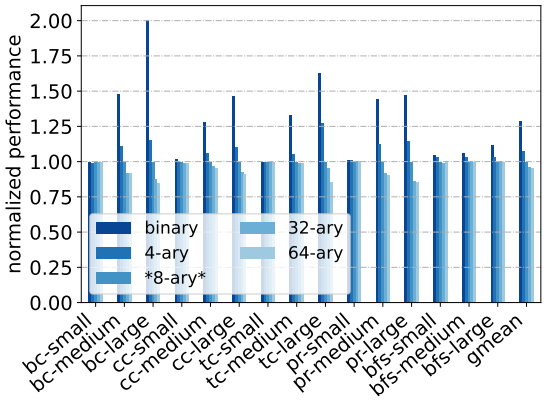


Figure 2.16: Normalized cycles to execute GAP benchmark suite for various integrity tree arities.

Consistent with other components of this analysis, higher arity trees provide significant performance improvements. When using a binary BMT, Fig. 2.16 shows a performance overhead of 29% relative to an 8-ary tree on average, and as much as a 2× overhead (`bc` large). Using a 64-ary tree can improve performance by 5% relative an 8-ary tree on average, and by as much as 16% in `bc` large. Ultimately, the takeaway from this analysis is that a higher arity tree improves performance.

Update Eagerness Fig. 2.12 shows the two possible implementation of integrity tree traversals for authentication. The reason an eager fetch can be performed is because a parent node address is computed as a function of the child address in a BMT (much like in an array-based min-heap). This thesis explores the impact of this decision explicitly, as the implementation proposed in Chapter 3 stores the parent pointer in the node itself, and must use a lazy fetch as a result. The analysis of this study is depicted in Fig. 2.17. A key takeaway is that, for these workloads, using a lazy tree fetch has a relatively negligible impact on performance. For most benchmarks in the suite, the performance difference in traversal schemes is less than 1%. However, in the worst case, the lazy fetch approach incurs an 11% overhead (`tc` large) relative to the eager fetch. The takeaway of this approach is that using a lazy fetch is not detrimental to performance.

2.5.5 State of the Art

The integrity tree is the primary source of performance overhead in secure memory. As such, its optimization has been the source of robust study.

Cache Block Utility Similar to encryption, several prior works have considered increasing the density of integrity trees to improve metadata cache utilization. These approaches will similarly increase the density of the integrity tree as encryption counters serve as the basis for integrity tree leaves. VAULT [256] proposes increasing the arity of an integrity tree closer to the root to reduce the global effective height fo the tree. Similarly, seeing as tree size is a function of memory size, CLFSIR [257] has proposed integrity trees for sub-regions of the address space.

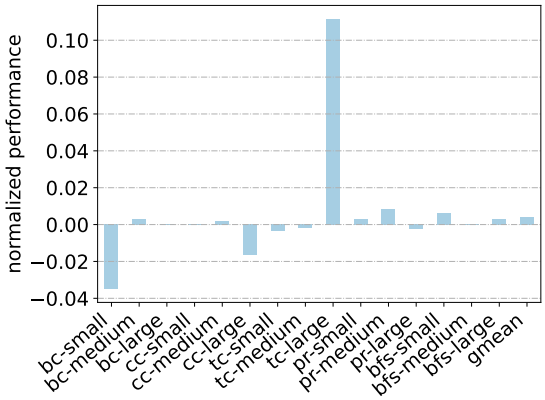


Figure 2.17: Overhead of lazy BMT traversal relative to eager traversal.

In the event that a path is not in the metadata cache, alternative approaches need to optimize the path to authenticate some data. ASSURE [214] proposes maintaining a dynamic, fast subtree to so that frequently accessed data can be authenticated against a subtree. Bo-tree [284] describes a small external tree that can protect a small percentage of frequently accessed data. It achieves this by modifying the leaf state to denote whether that data is to be authenticated against the fast or slow tree, and dynamically pages metadata in and out of the fast subtree as data becomes more frequently accessed.

Memory Hardware Co-Design Seeing as the MEE extends the memory controller logic, several works have considered co-designing integrity tree protocols with memory controller procedures. For instance, the memory controller is tasked with deploying logic for error correction codes (ECC). Seeing as ECC attempts to detect corruption, SYNERGY [225] proposed using these bits as a MAC so as to avoid having to also maintain an HMAC as metadata. Such an approach led to several similar follow on works [249, 304, 72, 70, 212].

Emerging memory technologies, such as non-volatile memories (NVMs), implement wear-leveling as an

additional (non-security) metadata field to improve device lifetime. ACME [252] leverages this notion to reduce the likelihood that an encryption counter will overflow, which allows for more aggressive reductions in bits for encryption counters. COVERT [254] and ASSET [253] build on top of this work to build such primitives. ExtraCC [51] combines the co-design of ECC, wear leveling, and the integrity tree for further optimization.

Several works have explored integrating speculation into the authentication procedure. PoisonIvy [151] describes a scheme in which the integrity tree authentication occurs off of the critical path of returning the data to the processor. If a hash mismatch is later found, a retroactive hardware panic is thrown before the execution can advance too far ahead. Several other works [91, 269, 235, 281] all similarly propose some speculative pre-computation on the MAC, tree path, or encrypted data to allow authentication to probabilistically remove computation off of the critical path of a data fetch.

To expand the likelihood of finding metadata on-chip, prior literature has explored improved caching schemes for security metadata in the metadata cache. In particular, MAPS [150] demonstrated that some security metadata exhibits irregular access patterns with long periods between reuse, but that it still exhibits locality. As such, HMT [230] and CTR+ [229] create hybrid caching protocols aware of this access pattern. Furthermore, Wang, et al. [286] propose a scheme in which the LLC can serve as a second level cache for the metadata cache to capture evicted encryption counters.

Fast Path Integrity Trees Seeing as authentication against the integrity tree is the primary performance bottleneck of secure memory, prior works have considered reducing integrity tree path lengths. While metadata caches achieve this, all associated performance benefits are closely tied to the cache utilization. An alternative approach would be to intentionally maintain intermediate nodes of the integrity tree as intermediate roots of trust. For example, ASSURE [214] is a protocol in which a single fast subtree is maintained in addition to the true integrity tree root. If the region in the fast subtree domain is accessed frequently, many authentications will not need to be performed up to the global root. Bonsai Merkle Forest [86] describes a protocol in which a set of subtree roots is determined dynamically at runtime. It does so by determining if a current subtree root is hot (in which that subtree root is moved towards the leaves) or cold (when it is moved upwards towards the root). Bo-tree [284] describes a protocol in which a small, auxiliary tree is maintained towards which data may dynamically be redirected.

Alternatively, the shape of the integrity tree itself may be skewed such that frequently accessed data has shorter authentication paths. Szefer, et al [255] propose a scheme in which the integrity tree shape is statically skewed. If software is aware of this architecture, the application or operating system can then prioritize allocation of frequently accessed data to these physical addresses. Several alternative schemes have proposed dynamically restructuring the integrity tree shape at runtime [276, 278, 277, 274, 183, 275, 182]. These works typically target embedded devices, where the memory size is small enough that the restructuring operations will not become the runtime bottleneck.

Avoiding Integrity Trees Instead of optimizing the integrity tree, some prior work have sought to optimize secure memory by avoiding the integrity tree altogether. Instead, a memory device may maintain a MAC along with data in memory to ensure its integrity to protect stored data at rest. Then, the device may develop an authenticated channel to a processor to ensure data is secured while in transit.

This approach typically considers profiling memory attacks and add defenses for these attacks. InvisiMem [4] describes the time access patterns to leak data from memory and puts mechanisms in place to defend against them. The motivation for this approach is defending against correlation attacks between consistent encrypted data states for far memory. In particular, this work proposes updating the encryption counter on memory reads as well as writes to eliminate replay attacks. SecDDR [82] defends against attacks that originate from the memory bus, so a replay-protected bus is sufficient to defend against memory-based replay attacks. To achieve this protection, SecDRR describes a protocol in which the memory controller and memory device both maintain a MAC of the data state.

As described in this chapter, memory devices are subject to a variety of corruption mechanisms. Sec. 2.2 shows how even relatively restricted attack surfaces can be exploited to perform powerful attacks. An integrity tree provides more robust protections to detect *arbitrary* corruption. As a result, the work in this dissertation improves integrity tree-based secure memory protocols.

Chapter 3

Cordelia: Using Huffmanized Merkle Trees for Scalable Secure Memory

3.1 Introduction

As demonstrated in the Sec. 2.3.2, secure memory comes at a steep performance cost as a byproduct of maintaining metadata. To optimize this protocol, state-of-the-art proposals and commodity secure memory hardware alike integrate a *metadata cache* [93, 150, 151, 13, 130] for recently accessed metadata. The benefits of this cache are twofold: ① metadata can be fetched at lower latency than going to memory, and ② the metadata cache resides on-chip, so cached values are trusted. That is, when authenticating some data, BMT nodes need to be fetched from the leaves up to the first metadata cache hit (or the BMT root if none of the path is in the cache). This chapter emphasizes the point that the benefit of ② is significant.

Unfortunately, secure memory has been deemed too slow for commodity processors. In the white paper promoting SGXv2, Intel notes: "Numerous studies have shown that workloads that make significant use of SGX memory have their performance impacted. The reason for this is the additional bandwidth required to fetch the various levels of the integrity tree protecting SGX memory" [130]. Given the benefits promised by the metadata cache, it would seem that hosting a larger metadata cache would resolve this issue. However, the processor chip area is very constrained, so very little space can be allocated on-chip to the metadata cache. In fact, it has been demonstrated that Intel SGXv1 makes use of a relatively small metadata cache (64kB [109]), which is comparable in size to a first-level data cache. At the same time, this metadata cache is tasked with capturing the locality of secure metadata at a memory access granularity (i.e., L3 misses, writebacks, etc). As a result, many workloads, such as machine learning (ML) and high performance compute (HPC), are still bound by the bandwidth requirement to implement secure memory [169, 312]. This trend will only continue to worsen as applications make increasing demands of memory devices [302, 95, 71, 115], and implies that metadata caches will suffer a *scalability* problem due to the fixed capacity of the cache. Furthermore, the cache's decreased utility will lead to further secure memory-driven performance degradation in memory-sensitive workloads.

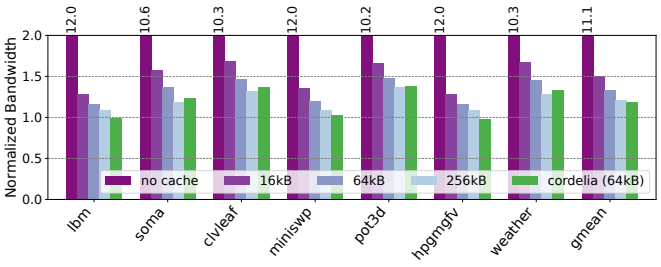


Figure 3.1: Bandwidth overhead of secure memory relative to a non-secure memory based on metadata cache size.

Fig. 3.1 demonstrates that the benefit of the metadata cache is significant. This analysis demonstrates the additional memory bandwidth consumed by secure memory compared to a non-secure memory with various metadata cache sizes (full evaluation specs are detailed in Sec. 3.4). Doing so models varying degrees of metadata cache resource constraint. When reducing the metadata cache from 64kB to 16kB, the average memory bandwidth consumed increases 20%. Conversely, increasing the metadata cache from 64kB to 256kB reduces the bandwidth overhead by 13%. This indicates that the benefits promised by the metadata cache are closely coupled with how efficiently it is utilized. In addition, the limited, fixed metadata cache size will suffer a *scalability* problem as workloads continue to become more memory intensive. Our approach, *Cordelia*, achieves comparable bandwidth to an unconstrained metadata cache.

In order to make secure memory suitable for commodity deployment, it is imperative to reduce the bandwidth requirements needed to implement secure memory. This chapter aims to achieve this by reducing the number of BMT accesses (and thus increasing the benefit of ② for the metadata cache listed above) for memory-intensive workload. The approach works by reducing the leaf-to-root path length for frequently accessed data. This means that, even if relevant BMT nodes are not in the metadata cache, authenticating data against the BMT root does not require as many metadata accesses.

The goal of this chapter is to *enhance* optimization ② of using a metadata cache. The metadata cache reduces BMT accesses for *recently* used metadata while the data structure reduces accesses for *frequently* accessed metadata, even when these addresses are not captured by temporal or spatial locality.

To do so, this chapter works from the key insight that the BMT faces performance limitations by not considering application behavior. The approach is to consider application behavior in the design of the data structure so that more frequently accessed data requires less work to authenticate. In particular, this chapter proposes re-organizing the tree such that frequently accessed addresses have shorter BMT leaf-to-root paths by leveraging *Huffman trees* [118], which maintain the provably shortest leaf-to-root path relative to access frequency. As a result, the optimization for shorter leaf-to-root authentication paths will be enforced by the data structure regardless of the cache utilization. Such an approach achieves benefit ② of the metadata cache without being subject to the limitations of fixed cache size.

Cordelia achieves adaptability by using the Faller-Gallager-Knuth (FGK) [83, 90, 139, 279] adaptable Huffman tree algorithm. This design decision is important, as using the construction method in [118] works best with *a priori* knowledge of global access frequencies and that these access frequencies would be static, which does not apply to the secure memory problem. For this reason, Cordelia adapts the FGK and BMT structures to be safely integrated with one another. Consequently, Cordelia improves performance at runtime by reducing the number of memory accesses associated with frequently accessed addresses. This alleviates the bandwidth requirement needed for secure memory.

The contributions of this chapter are as follows:

1. We propose *Cordelia*, a secure memory architecture whose integrity tree is structured as a dynamic, Huffmanized Merkle Tree to reduce metadata accesses in secure memory.

2. We adapt the classical FGK dynamic Huffman tree algorithm to enable dynamic restructuring of the BMT.

3. Our evaluation demonstrates the scalability limitations of the metadata cache and shows that Cordelia benefits performance by 22% relative to the state-of-the-art and 31% relative to commodity secure memory on average and by up to 55% for HPC workloads.

3.2 Background and Related Work

Much of the relevant background concerning secure memory is covered in Chapter 2. This section in this chapter primarily concerns the relevant background on Huffman trees and details the relevant prior literature in this space.

3.2.1 Huffman Trees

A Huffman tree [118] is a binary tree structure where path lengths are proportional to the frequency with which leaves are accessed. The more frequently that a node is accessed, the closer it will be to the root. If one particular leaf node is accessed more frequently, then it will have a provably equivalently shorter path from leaf to root the shorter its path through the tree, which minimizes the average path length through the tree relative to access rate.

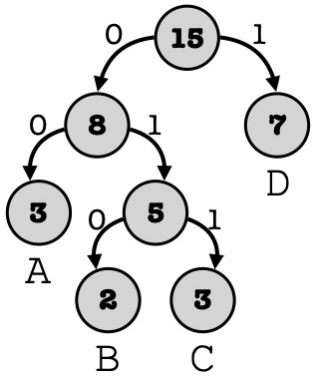


Figure 3.2: Example for Huffman encoding. A Huffman tree is constructed from the relative frequency of each character. To encode `A`, the tree is traversed left twice so the resulting code is `00`. `B` is encoded as `010`, `C` as `011`, and `D` as `1`.

The benefits of Huffman trees have been studied in a variety of contexts. Huffman trees are often applied to encoding information [187, 279, 139, 90, 110]. In particular, a Huffman encoding scheme is the provably minimal information required to represent data relative to the frequency with which the tokens comprising that data are used. This may be useful when attempting to perform image compression [242], energy/power reduction [174, 198], DNA compression [201], etc. To do so, a Huffman code for some data is derived by constructing a Huffman tree over some set of tokens comprising the leaves of the tree and concatenating the code of the tokens. The code of token is derived by finding its path through the tree from the root to the leaves: store 0 when going left and 1 when going right. The process is depicted in Fig. 3.2.

The tree shape of this Huffman tree algorithm is a function of its construction. Therefore, if the tree shape ever needs to change, the tree must be reconstructed. This is a relatively safe assumption in certain contexts where the distributions are statically known; for example, the relative usage of characters in an

Listing 3.1: The construction of a Huffman tree.

```
// T is a declared/defined type
// Node is a declared/defined type that maintains a data field,
// a frequency count, and a reference to its left/right children
priorty_queue<T> pq;
set<Node<T>> leaves;

// sorted PQ of leaves from least to greatest
for (leaf: leaves) {
    pq.push(leaf);
}

/* Huffman tree construction */
while (pq.size() > 1) {
    // pop the two least frequently accessed nodes from the queue
    Node<T> n1 = pq.pop();
    Node<T> n2 = pq.pop();

    // create a parent node from the children
    Node<T> parent = { n1.getFrequency() + n2.getFrequency() };
    parent.left, parent.right = n1, n2;

    // add the new parent to the priority queue
    pq.push(parent);
}

Node<T> huffman_tree_root = pq.pop();
```

alphabet are relatively stable when encoding strings of language. However, adapting the tree shape requires total tree reconstruction if the context isn't known in advance.

Consider applying a Huffman tree to secure memory. As established in Chapter 2, the integrity tree is the bottleneck of a secure memory due to the large number of fetches. Seeing as the benefits of Huffman coding are a function of short path lengths, performance benefits may follow applying this construction to an integrity tree. As a result, Huffman trees provide an interesting framework for reconsidering BMT structures for secure memory, but there are several challenges to directly applying the encoding procedure in [118] to secure memory. For one, a Huffman tree assumes that the access frequencies are globally known *a priori* and static throughout the execution of an application. Such an assumption is unreasonable in the context of memory hardware: applications are highly dynamic in their behavior and memory devices operate agnostic to the behavior of any individual application. To counteract this, one could imagine reconstructing the Huffman tree from time to time. Unfortunately, the $\Theta(n \log n)$ cost to construct a Huffman tree is an arduous operation when n refers to gigabytes or terabytes of physical addresses.

Given this observation, this dissertation considers alternative constructions of Huffman trees. In particular, it focuses on *adaptive* protocols, such as the FGK algorithm [83, 90, 139, 279]. The FGK algorithm is based on the characterization of a Huffman tree (described in [139, 279]), in which nodes are strategically organized such that weights are sorted from least to greatest as nodes are scanned from the bottom left to the right and upwards (i.e., $\rightarrow$, $\uparrow$, $\rightarrow$, etc). Given this configuration, incrementing and adjusting the shape of the tree is achieved with significantly less work per operation as the algorithm takes into account the shape of the prior distribution. The algorithm is described formally by Vitter in [279].

For clarity, the update protocol is overviewed with an example, highlighted in Fig. 3.3. To update (increment) the weight of a leaf node, FGK calls an `update` operation that applies the following iterative

modifications to the tree to retain it as a Huffman tree: starting with the leaf as the current node (i.e., the red dotted node in Fig. 3.3a), the subtree of the current node N is "*interchanged*" with the subtree of the highest "ordered" node of the same weight (i.e., the node that is furthest "up and to the right" of the same frequency, shown in Fig. 3.3b). After interchanging, the weight of node N is incremented. Notice that, because N was the "highest order node" of its frequency prior to incrementing, the arrangement of nodes by frequency is preserved. Then, the process is repeated with the parent of node N becoming the new current node (i.e., Fig. 3.3c). The procedure stops after incrementing the root's weight. Each interchange (which trivially maintains the above Huffman tree characterization) allows the subsequent increment to operate on the highest ordered node of its weight so the increment does not violate the characterization, and thus the resulting tree is a Huffman tree.

Although the FGK adaptive algorithm provides a strong basis to reimagine the shape of a BMT, doing so naïvely faces several key pitfalls that are addressed in this chapter. For one, the original conception of FGK describes a protocol in which the tree shape adapts to *any* change to the distribution of accesses. In the context of secure memory, this means any memory access could result in tree reconstruction. Furthermore, the FGK algorithm is a fundamentally sequential algorithm, and applying sequential algorithms to parallel environments (such as memory devices) can lead to consistency violations. Finally, swapping pointers and interchanging nodes may lead to replayable states, which presents a new attack surface for malicious adversaries. Sec. 3.3 describes the explicit steps that Cordelia takes to address these pitfalls in applying the FGK algorithm to secure memory.

3.2.2 Related Work

A few works have looked into Huffmanized Merkle trees for secure memory [316, 182, 183, 277]. Unfortunately, the feasibility of applying these works to full memory protection of systems outside the edge (i.e., IoT or FPGA) has been limited in practice for a variety of reasons. These devices tend to have smaller capacity, and the associated integrity tree is small. This work targets secure memory for commodity devices with memories with at least gigabytes of capacity and large integrity trees as a result.

FAST [316] describes a statically Huffmanized Merkle tree for secure FPGA memory. In particular, this work proposes a binary Merkle tree in which some skew can be generated. The work provides an insight into how the tree must be organized in order to have a dynamic shape by classifying nodes as encryption counters (i.e., BMT leaves), Merkle tree leaves (i.e., parents of encryption counters in a BMT), and inner tree nodes. Merkle tree nodes need to maintain a reference to the parent node. The approach uses a modified version of the Huffman tree construction proposed in [118], which limits the dynamism of the approach. Furthermore,

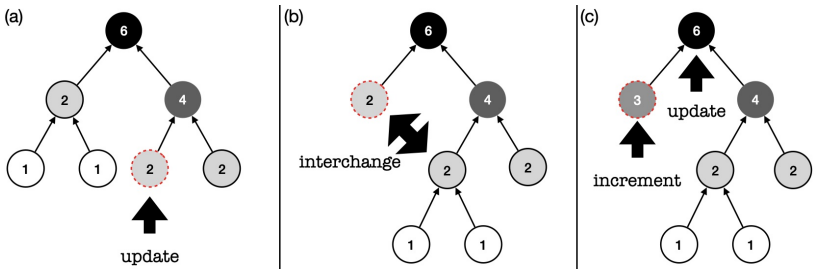


Figure 3.3: Sample call to `update` in an FGK Huffman tree.

the modifications to the construction fell its theoretical optimality.

Millar, et al [183, 182] implements several variants of the Pigeon and Begnio dynamic Huffman tree [207]. To do so, a reference needs to be maintained to parent nodes, uncle nodes, and sibling nodes in the node. This increases the memory overhead of the integrity tree structure by 3×. However, this work does not consider the implications of parallel updates to the tree shape and authentications. Without this safety mechanism in place, the hashes in the authentication path may not correspond to the current child node, which would lead to a hash mismatch. Furthermore, the analysis in this work indicates that such an approach *hurts* performance relative to static Merkle trees for secure memory.

Vig, et al [273, 277, 274] proposes using the dynamic Huffman construction scheme proposed by Pigeon and Bengio [207] as the basis of the Merkle tree for secure memory. In this scheme, the frequency of access is maintained in an look-up table stored in memory. By doing so, frequency counters are not maintained in the integrity tree nodes to ensure that maximal space in the node is allocated for security metadata. The scheme dynamically updates the tree shape on every memory access. Unfortunately, the binary nature of the tree means that the BMT will not be as dense and metadata cache performance will degrade as a result. Additionally, in order to safely update the tree shape, authentications are blocked while updates occur. Furthermore, the frequency of tree shape update quickly becomes the bottleneck in performance. CaDST [275] is a follow on work from Vig, et al that shows that the benefit of using Huffman trees augments the benefits of a metadata cache, and using a metadata cache drives significant performance benefits in authentication.

This chapter presents Cordelia, a Huffmanized Merkle Tree for secure memory that addresses the limitations of the prior work. Cordelia explicitly addresses each of these challenges, making it practical for deployment. Furthermore, Cordelia also looks into the scalability of its approach by considering various schemes for incrementing counters and freezing levels out of the Huffman region.

3.3 Cordelia

Thsi section describes the design of *Cordelia*, a Huffmanized Merkle Tree for secure memory. To implement such an approach, Cordelia applies a modified FGK adaptive Huffman tree algorithm [279, 83, 90, 139] to the BMT in secure memory. In addition to lightweight adaptations to implement an 8-ary (standard for secure memory) FGK tree, this work observes three non-trivial design features, each of which imposes implementation and/or security challenges.

Observation 1: Integrity tree nodes need to maintain a reference to their parent to implement dynamically adaptable paths.

In order to account for this observation, Cordelia must modify the layout of tree nodes in order to store a reference to their parent. Furthermore, the reference should not provide a new attack surface; any corruption of the reference should be detectable by the authentication procedure. Sec. 3.3.1 discusses how, if done naïvely, such an attack is possible and how Cordelia is resilient to this attack.

Observation 2: Implementing a Huffman tree requires additional ordering metadata, whose storage must also be considered.

It is insufficient to store frequency weights *in tree nodes*, as doing so would occupy space that could otherwise be used to store HMACs or counters. Furthermore, frequency counters could be subject to overflow in the original conception of FGK. Such a behavior was an out-of-scope consideration in the theoretical instantiation, but could have detrimental performance impacts in real hardware deployment. To account for this, Cordelia proposes a new mechanism to track access frequency counters and Huffman weights that explicitly addresses these pitfalls. Sec. 3.3.1 and Sec. 3.3.2 describe how this is achieved in Cordelia.

Observation 3: The FGK algorithm is a sequential algorithm, but memory devices are highly parallel.

Cordelia implements mechanisms to ensure that the tree shape can safely change without paths changing mid-authentication. Furthermore, this safety mechanism allows for parallel memory accesses and authentications in the device while still having an adaptive tree shape. Sec. 3.3.3 describes how Cordelia achieves this.

3.3.1 Pointer-Chasing Tree Nodes

This section first describes how Cordelia accounts for the challenges imposed by Observation 1. In prior BMT-based secure memory schemes, the address of the data and metadata is used to compute the address of the respective encryption counter and parent node [222, 62, 99]. However, in order to have dynamic paths, BMT nodes in Cordelia need to maintain an explicit reference to their parent, as the parent node may change in calls to `update`. The MEE still finds the encryption counter for some data using its address, but all parent nodes above the encryption counters are found by dereferencing a pointer. As such, the parent of all tree nodes must be stored alongside the node's data (i.e., *pointer-chasing*). In order to make space to store pointers in BMT nodes, space must be "borrowed" from elsewhere in the node to remain word-aligned.

In Fig. 3.4 highlights how space is allocated to BMT nodes in Cordelia. Recall that per-page encryption counters are composed of sixty-four 7-bit minor counters and one 8-byte major counter in counter-mode encryption. Cordelia's nodes need 8 bytes of space and makes this space by reducing the minor counters to 6 bits. To account for this change, Cordelia assumes the most significant bit of this value to be 0 in the minor counter field of the AES engine input for encryption/decryption. The 64 bits that are freed as a result of this change are used to store the encryption counter's parent pointer field (see Fig. 3.4). Inner BMT nodes in Cordelia are composed of eight 7-byte hashes to make space for the parent pointer. These structural modifications are intentionally lightweight by design in accordance with Observation 2.

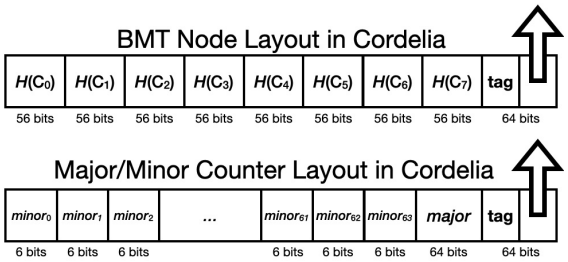


Figure 3.4: Layout of counters and inner nodes in BMTs to make space for 32 bit tag and 32 bit pointer in Cordelia.

Note that the parent reference in Cordelia nodes must be *tagged* with the original node index in the integrity tree to remain robust against replay attacks. To highlight how such an attack could occur, suppose two different nodes in the tree have identical metadata states, as demonstrated in the left side of Fig. 3.5. Such a scenario implies that interchanging these two different metadata values could be legitimately verified by the same parent. This primitive is dangerous; if different metadata values can be authenticated by

the same parent node, the MEE cannot distinguish between legitimate, uncorrupted metadata from maliciously corrupted values. Thus, Cordelia tags the pointers with the original value to ensure that similar data always has a unique identifier per tree node. This is demonstrated on the right side of Fig. 3.5.

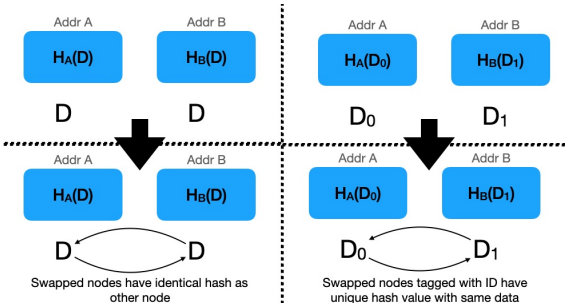


Figure 3.5: Cordelia tags pointers with a unique ID to ensure different nodes with the same value produce different values.

3.3.2 Frequency Tracking

Cordelia maintains access frequencies in an auxiliary structure outside of the central BMT. As conveyed in Observation 2, storing frequency counters in BMT nodes themselves would further restrict the space per node allocated to security metadata. Thus, only the parent pointer described in Sec. 3.3.1 comes at the expense of additional security metadata requirements. Furthermore, making this design decision naturally allows Cordelia to relax the precise definition of access frequency to the rate at which some node is accessed as a function of n. Sec. 3.3.3 describes the mechanism of this below, and overview the benefits of this decision.

The organization for the proposed scheme is presented in Fig. 3.6. In the implementation of FGK, a call to `update` (i.e., to reorganize the tree shape) is called whenever a leaf has been accessed n times. Thus, for each leaf, Cordelia maintains a set of $\log_2 n$ bit frequency counters that, on overflow, dictate when to call the `update` procedure. In addition to these counters, these frequency counters also maintain a reference to an auxiliary structure that maintains the *weight classes* for all nodes in the Huffman tree. These weights are tracked in a "weight class list," where elements in the list maintain: ① the weight as a 64-bit counter; ② a reference to the highest order tree node of that weight; and ③ a reference to the next element of the weight class list.

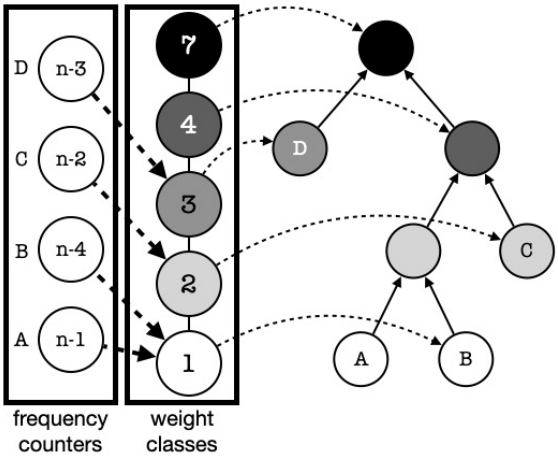


Figure 3.6: The organization of FGK frequency tracking structures in Cordelia.

In Cordelia, calls to FGK `update` entail performing interchanges to update the tree path, and are consistent with the protocol in [139], but incrementing node weights is achieved through the weight class list. In particular, incrementing a node with order o from weight w to $w+1$ means adjusting the reference to weight w to $o-1$. If $w+1$ is not currently in the weight class list, then an entry needs to be allocated and added to the list. Conversely, if o was the only member of w, then w is removed from the weight class list and can later be repurposed for new weight classes.

Although the number of possible weights is infinite, this work observes that the number of *active* weights is equal to the number of nodes in the Huffman tree. As such, the amount of space to reserve for the weight

Listing 3.2: Management of `free_list` for frequency weights.

```
struct Node<T> {
    T data;
    Node<T> *next;
};

Node<T> head;
Node<T> tail;

Node<T> allocate() {
    Node<T> n = head;
    head = head->next; // does not require a memory operation
    return n;
}

void reclaim(Node<T> n) {
    // note: we don't need to invalidate n's next pointer
    tail->next = n; // requires memory operation
    tail = tail->next; // does not require a memory operation
}
```

class list is proportional to the tree size. Sec. 3.3.4 describes a worst case mitigation mechanism for this behavior.

3.3.3 Concurrency for Secure FGK

To achieve this, Cordelia implements a modified *reader-writer lock* (RW lock) [117] in the MEE logic as a concurrency control mechanism. RW locking allows for multiple readers to access the shared state in parallel whereas a writer must be the sole actor that can access the state. As such, updates to shared state are always viewed consistently. In Cordelia, normal memory requests (reads that need authentications and writes that require metadata updates) are considered "readers" because they maintain the BMT shape. On the other hand, requests that implement the FGK protocol are considered "writers" as they modify the tree shape.

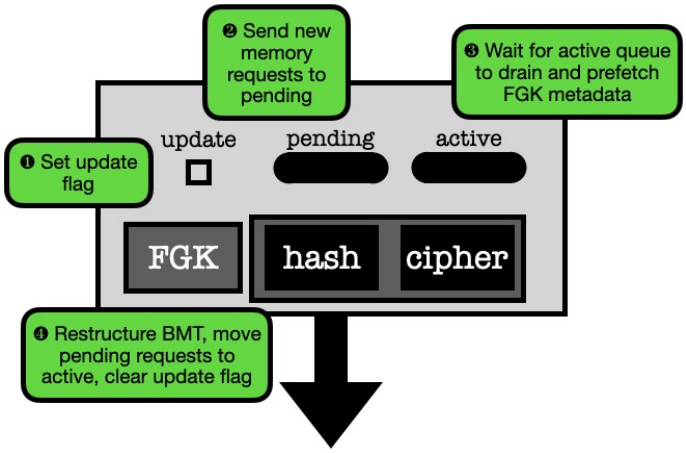


Figure 3.7: Huffman and non-Huffman regions of a tree with 1 frozen level. Red nodes are "dynamic" and blue nodes are "static."

As depicted in Fig. 3.7, the lock is implemented as two queues, for "active" and "pending" requests, and an "update" flag. If the FGK engine decides to restructure the tree shape, it sets the "update" flag. When this flag is set, incoming memory requests from the processor are buffered in the "pending" queue. Once the last request is processed from the "active" queue, the FGK engine issues the requests to update the tree pointers and restructure its shape. Note, the FGK metadata state can be fetched and the updated state

can be computed while waiting for the "active" queue to drain. After the FGK updates have responded, the processor requests can be moved from the "pending" queue to the "active" queue, and the "update" flag can be cleared so that normal execution can resume.

Unfortunately, using this concurrency control can quickly degrade performance. If every memory access calls `update`, then utilizing the RW lock turns Cordelia into a blocking memory. To address this, Cordelia relaxes the notion of "access frequency" as described in Sec. 3.3.2. Relaxing this definition has the effect of fewer memory accesses triggering the RW lock to block authentications (where the application advances). Consequently, the memory device is able to fetch and authenticate several data requests in parallel. Spending time with the "update" flag set can still lead to pauses in application fetches; however, when using 64 as a default value for n in practice, the typical blocking case only incurs approximately 10-25 memory accesses to re-balance the tree and that the cost of restructuring the shape is amortized by the benefit of fewer metadata accesses. In the worst case, however, the amount of time restructuring the tree is a function of the longest path of Huffman tree nodes. Mitigation strategies of the worst-case behavior are discussed in Sec. 3.3.4.

3.3.4 Worst Case and Spatial Overhead Mitigation

Both the size of Cordelia's auxiliary structures and the worst case bound for time spent blocking is bound by the number of nodes in the Huffman tree. To mitigate this, this work proposes a scheme in which it relaxes the granularity with which Cordelia tracks access frequencies by "freezing levels." That is, this work defines various regions of the tree as being "static" and other regions as "dynamic." Fig. 3.8 depicts the organization of these regions. The size of the static region is configurable and can be set by the end-user in the BIOS. This approach is inspired by Awad, et al [24], which proposes a similar tree partitioning mechanism for crash consistency models.

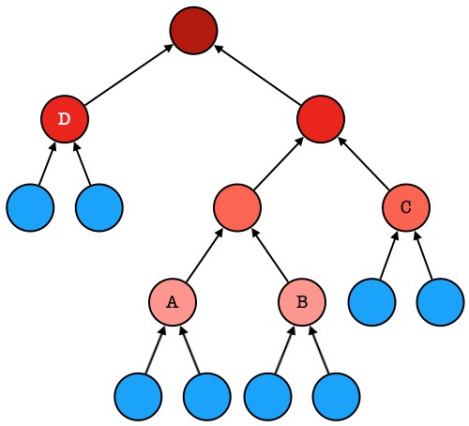


Figure 3.8: Huffman and non-Huffman regions of the BMT with 1 frozen level. Red nodes are "dynamic" and blue nodes are "static."

The space that Cordelia reserves for frequency counters is a function of the number of Huffman leaves in the tree. Given the fan-out property of the 8-ary BMT, "freezing" n lowest levels of the BMT out of the Huffman region decreases the number of Huffman nodes by approximately a factor of 8^n. Seeing as these are the lower levels of the Huffman tree, the number of leaves is similarly reduced by 8^n. Reducing the number of leaves means that less space needs to be reserved for the frequency counters and reducing the number of Huffman nodes reduces the amount of space to be reserved to maintain the weight classes (depicted in Fig. 3.6). In addition, reducing the number of Huffman nodes reduces the longest potential path, which in turn reduces the worst case pause in execution during rebalancing. Sec. 3.4.3 details the precise spatial overheads of Cordelia in and details the effect "freezing levels" have on this overhead.

Partitioning the BMT into Huffman and non-Huffman regions implies that "Huffman" leaves essentially serve as roots over a static subtree. That is, if Cordelia is configured with more "frozen levels," then the

Table 3.1: Evaluation configuration for Cordelia evaluation.

On-Chip Configuration	
Processor	4 cores, x86 ISA, out-of-order 1GHz clock, 1 thread/core
L1 cache	48kB icache, 32kB dcache 2-cycle latency, 64B/block
L2 cache	512kB, 20-cycle latency, 64B/block
L3 cache	8MB, 32-cycle latency, 64B/block
Security Configuration	
BMT	8-ary tree nodes, 64-ary counters
Metadata Cache	64kB, 2-cycle latency
DDR4-based Memory Configuration	
Capacity	64GB, dual channel
Latency	150ns [188], 3200 MT/s [136]

resulting tree is more balanced. Consequently, mitigating the worst case behavior does so at the expense of the precision at which access frequencies can be tracked. In practice, however, this work observes that less granular views of access frequency do not negatively affect Cordelia and, in some cases, actually benefit performance. Sec. 3.4.3 discusses this in more detail.

3.4 Evaluation

In this work, Cordelia is implemented in gem5 [167] and evaluated with the hardware configuration specified in Table 3.1. This evaluation aims to show: ① the scalability limitations of commodity secure memory deployments (i.e., the approach used in Intel SGX [99]); ② the benefit of Cordelia relative to the state-of-the-art; and ③ a sensitivity analysis to highlight the impact of Huffman reconstruction on performance. This is done through implementation and evaluation three prior arts for comparison: ASSURE [214], VAULT [256], and Morphable Counters [224]. In general, reference to the "baseline" approach implies the commodity secure memory deployment from [99]. This approach allows us to evaluate the impact of details concerning secure memory on end-to-end application performance. This section also analyzes the spatial overhead of Cordelia and perform a security analysis of the approach. For the purposes of the evaluation, the implementation of the commodity secure memory deployment refers to the approach described in [99] as the "baseline" approach.

This evaluation is performed using three types of applications: (1) high performance computing workloads from the SPEChpc 2021 benchmark suite [71], (2) the breadth-first search (BFS) workload from graph500 [251] for graphs of varying sizes, and (3) a random access microbenchmark (MB), in which random indices are accessed in a large array, to demonstrate memory intensive behavior. In using the SPEChpc benchmarks, the architecture consists of a 2MB LLC and the "tiny" workload to demonstrate the efficacy of the approach. This is the largest benchmark that could be simulated in gem5 given the system requirements of the benchmark, but the LLC is shrunk to emulate the increased memory demand of the "small" workloads. The simulated the region of interest as defined by the source in the benchmark suite. The evaluation of graph500 is modeled off of prior work [251] by executing four iterations of BFS and turning off validation. Finally, the MB workload pre-allocates an array, and one billion accesses are measured to random indices of a large array as the region of interest.

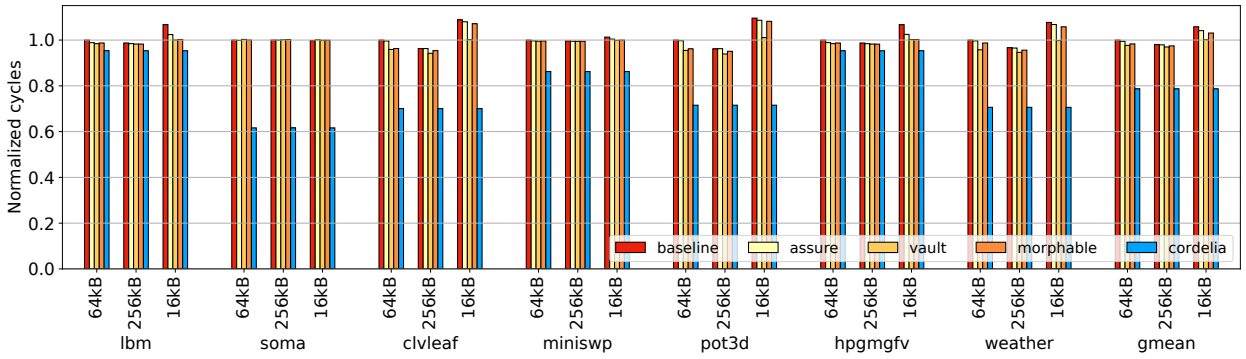


Figure 3.9: Cycles to execute the region of interest in each of the SPEChpc 2021 benchmarks. Results are normalized to the baseline with a commodity metadata cache. Lower is better.

3.4.1 Performance

The evaluation of the end-to-end runtime of the SPEChpc 2021 benchmark suite is shown in Fig. 3.9. The figure shows several metadata cache configurations of both the baseline secure memory protocol and Cordelia to highlight its scalability. In particular, the evaluation entails a "large" metadata cache of 256kB and a "small" metadata cache of 4kB in addition to the metadata cache in a commodity deployment (i.e., 64kB [109]). By using these configurations, this evaluation demonstrates the impact of the metadata cache on performance. That is, a smaller metadata cache demonstrates what the expected behavior should be when the cache is more constrained by the active set of metadata.

As demonstrated by the figure, Cordelia improves performance by 17% on average across the benchmark suite with a commodity secure memory configuration. In addition, the performance in the baseline is particularly sensitive to metadata cache size. Under these workloads, a smaller metadata cache can lead to performance degradation up to 14% (`pot3d`). On the other hand, Cordelia's performance optimization is agnostic to the metadata cache in these workloads. Even with a small metadata cache, Cordelia's performance is impacted by less than 1%. With a small metadata cache, Cordelia improves performance by 31% on average and by up to 55% (`clvleaf`) relative to the baseline. Cordelia also improves performance by 26% relative to ASSURE, 21% relative to VAULT, and by 24% relative to Morphable Counters. Cordelia is able to reduce path lengths more than ASSURE, in which the subtree may only move horizontally. Furthermore, despite the improved utility per metadata cache block in VAULT and Morphable Counters, Cordelia enhances the reduced authentication path length optimization beyond the metadata cache alone.

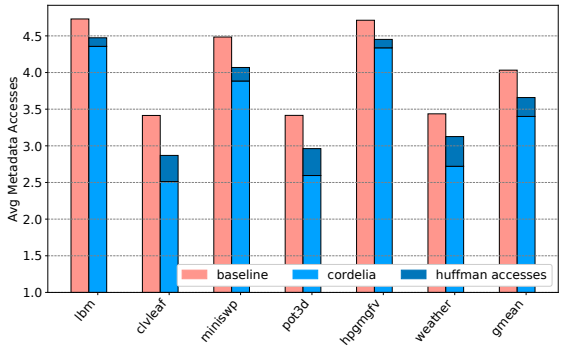


Figure 3.10: Breakdown of average memory accesses per data access in each of the SPEChpc 2021 benchmarks. Fetches described by metadata type.

The benefits of Cordelia can be attributed to the reduced work to authenticate data. Prior work has demonstrated that secure memory is a performance bottleneck in memory-bound applications due to the

bandwidth requirements to fetch metadata [169]. To demonstrate this, Fig. 3.10 shows the number of memory fetches for an average authentication required with commodity cache sizes for each of the SPEChpc benchmarks. These are broken down by the access for the application data, the authentication metadata (i.e., counter, HMAC, and BMT nodes), and the work to restructure the Huffman tree in Cordelia. The figure shows that, across workloads, BMT fetches are the majority of device accesses. Furthermore, Cordelia reduces the number of memory accesses by 14% on average across the suite as compared to the baseline. This is despite the additional work to implement and maintain the Huffman tree, which accounts for 8% of the total memory fetches on average. This evaluation finds that this reduction in fetches can benefit performance to the extent that these workloads are bound by memory fetches. That is, the performance benefit is greater for workloads (such as `clvleaf` and `pot3d`) in which there is a significant reduction in additional fetches for metadata (16% and 18%, respectively) and the application is highly memory intensive.

To study this further, the baseline secure memory protocol is modeled with a memory-intensive random access microbenchmark while controlling for metadata cache size. The motivation for this is to maximize the percentage of execution that is bound by the memory system. The results are shown in Fig. 3.11 and further highlight the benefits of Cordelia, where up to 22% performance improvement relative to a baseline protocol with a commodity cache, and by up to 31% with a small metadata cache. The benefit of the metadata cache decreases as the memory size increases. When performing random access to a 1GB array, a larger cache improves performance by 10% as compared to a 3% improvement with a 4GB array. In addition, this evaluation shows that Cordelia performs better for smaller workloads. Using the 1GB workload, Cordelia improves performance by 21% relative to the baseline as compared to 12% in the 4GB workload. This can be attributed to the fact that random access serves as an adversarial deployment for Cordelia, as there is little re-use by definition. Smaller arrays imply that the resulting Huffman tree will have more skew, and have a greater benefit on end-to-end performance as a result.

3.4.2 Metadata Cache Scalability

The results from Sec. 3.4.1 calls into question the scalability of the metadata cache. That is, as pressure on the memory device increases due to bandwidth sensitivity in the application, the utilization of the metadata cache (and the benefits that it can provide) decrease. Furthermore, HPC workloads with larger memory footprints than could be modeled in gem5 are likely to have a larger set of active metadata than can be supported in the metadata cache.

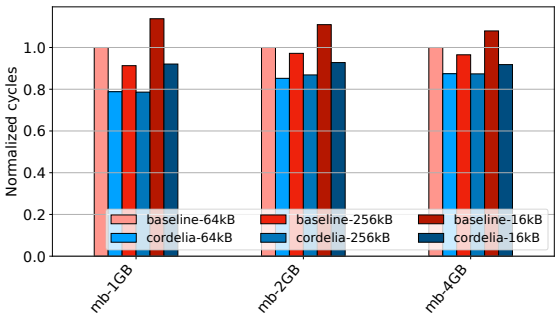


Figure 3.11: Cycles to execute 1 billion accesses to various sized arrays. Results are normalized to the baseline with a commodity metadata cache. Lower is better.

This analysis shows that a more constrained metadata cache directly correlates to degradation of end-to-end performance. In fact, there is a 20% slowdown simply by using a 4kB metadata cache as compared to a 64kB metadata cache in both the 2GB and 4GB random access microbenchmark. This implies that, although certain prior literature that increases BMT density [224, 256] may help to a

point, increasing memory footprints will continue to put pressure on the metadata cache and performance will suffer as a result.

Given the observation of limited metadata cache scalability, the evaluation includes a sensitivity analysis of Cordelia as compared to the baseline secure memory protocol with respect to metadata cache size. This analysis, compares the baseline secure memory protocol in [99] to Cordelia. The performance of Cordelia is normalized to the performance of the baseline secure memory protocol in Fig. 3.13 to show the relative speedup in each scenario.

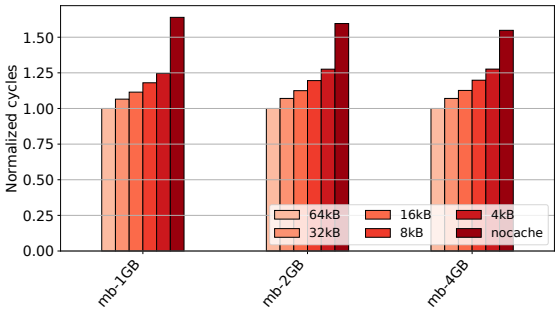


Figure 3.12: Normalized cycles to execute the baseline secure memory protocol to execute a random access microbenchmark for various metadata cache sizes.

This evaluation shows that, in cases where the metadata cache is constrained (i.e., smaller metadata cache sizes), Cordelia can have significant performance benefits. For example, in both the 2GB and 4GB random access microbenchmark, the evaluation finds that Cordelia can improve performance by as much as 29% compared to a 4kB metadata cache while using the baseline secure memory configuration [99]. This implies that Cordelia can help recover the potential loss in performance in these constrained cases. Beyond this, the evaluation finds that the benefit of Cordelia scales more as the size of the workload increases. This is attributed to the fact that smaller metadata caches are less effective in reducing the effective metadata required per authentication. On the other hand, Cordelia reduces metadata access requirements *in the data structure*, which is agnostic to the application demands for the metadata cache.

The relative benefit of using Cordelia with a small cache in the 4GB workload is greater than that of the 1GB workload. Seeing as the utilization of a smaller cache is worse for the bigger workload, the relative benefit of Cordelia, a cache-agnostic protocol, improves by comparison. By contrast, a smaller cache will not negatively impact performance to the same extent in the 1GB workload.

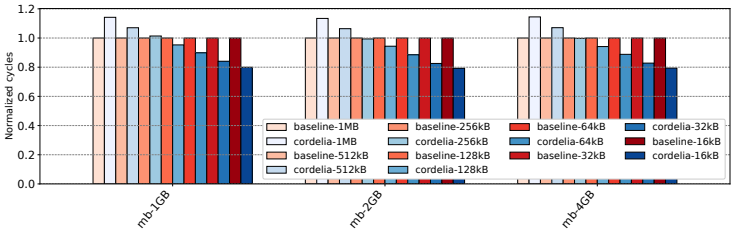


Figure 3.13: Cycles to execute microbenchmark for various metadata cache sizes. Performance normalized to baseline for equivalent cache.

Note, this observation is most evident in the microbenchmark as it puts the most stress on the memory system. While the SPEChpc workloads are memory bound, there is enough computation oriented execution to amortize the cost of Huffman reconstructions. On the other hand, when the application is entirely memory bound and the metadata cache is sufficiently large to efficiently optimize secure memory, the overhead of Huffman reconstruction becomes evident relative to the baseline secure memory case. In Fig. 3.11, this tends to happen with a 512kB metadata cache. However, as soon as metadata cache performance begins to deteriorate, the benefits of having shorter authentication paths becomes evident.

3.4.3 Impact of Worst Case Mitigation

As described in Sec. 3.3.4, the worst case behaviors in Cordelia are largely a function of the number of nodes in the Huffman region of the tree. Freezing the f lowest levels of the Huffman tree can remedy some of these behaviors, but the impact of this decision on performance is non-trivial. In particular, freezing the f lowest levels of the BMT out of the Huffman tree essentially trades off the precision at which Cordelia can track access frequencies with the spatial overhead required to perform such tracking. Surprisingly, there can be positive implications of freezing levels on performance beyond the theoretical case.

Fig. 3.14 compares the normalized cycles to run the graph500 BFS workloads in Cordelia without a metadata cache using different levels for f to highlight the impact of this design decision. As demonstrated by the evaluation, increasing f can have a positive impact on performance to a point, especially for workloads with larger memory footprints. This evaluation finds that, by reducing the number of Huffman nodes, the amount of work for the FGK algorithm to do is reduced. That is, when applications access memory more uniformly, the resulting Huffman tree is more balanced. Thus, increased precision does not benefit performance in these cases. Consequently, reducing the number of Huffman nodes from Cordelia reduces the amount of work per call to `increment` in the FGK algorithm. This means that Cordelia can spend a greater percentage of time performing authentication. On the other hand, if too much precision is lost (i.e., $f = 6$ while performing BFS in a 2GB graph), then there can be a regression in performance as Cordelia cannot create enough skew to benefit performance.

3.4.4 Spatial Overhead

Cordelia maintains two auxiliary structures in order to implement the FGK algorithm: ① the frequency counters associated with each leaf and ② the list of node classes. Each of these structures requires an explicitly reserved region of physical memory where they are stored, and their size is a function of the number of nodes in the Huffman tree.

To implement relaxed updates in the FGK part of the tree, Cordelia tracks how many times a particular Huffman leaf has been updated between 0 and n, and maintains a reference to that leaf's associated weight class. By default, this work assumes n to be 64 (i.e., 6 bits of storage), and as described below there are t possible weight classes where t is approximately 300,000 for a 64GB memory (i.e., 19 bits of storage). Thus, each of the 262144 frequency counters in a 64GB can be expressed as an 4-byte field, resulting in 1MB to store the frequency counters.

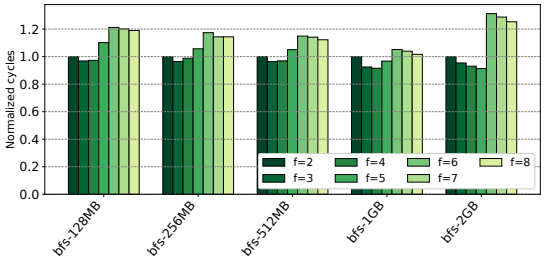


Figure 3.14: Sensitivity of number of frozen levels, f, while performing BFS in graph500.

Elements in the node class list maintain a reference to the highest order node in the weight class, a reference to the next greatest weight in the node class list, and a counter to maintain the current frequency for that class. The number of elements in the weight class list is equivalent to the number of Huffman nodes to account for the worst case (i.e., each node resides in its own weight class). Seeing as there are approximately 300,000 Huffman nodes in a 64GB memory, this means that each of these references can be expressed as 19-bit values. In order to ensure that elements can

be neatly word aligned, 90 bits are stored for the frequency counter. Thus, the size of each element in the weight class list is 16-bytes, and the total size of the list is approximately 4.57MB.

As described in Sec. 3.3.4, the spatial overhead of the auxiliary structures in FGK are largely a function of the number of Huffman leaves in the tree. By freezing the f lowest levels of the Huffman tree, Cordelia can reduce t by a factor of 8^{f-1} given the 8-ary nature of the BMT. That is, merely by setting f to 1 reduces the spatial overhead of Cordelia from 5.57MB to 714kB, setting $f = 2$ requires 90kB, etc.

3.4.5 Security Analysis

The methodology for the security analysis is based on prior secure memory literature [224, 256]. These serve as the standard methodology in analyzing the security of this literature. In particular, the analysis is predicated on the fact that the secure memory protocol is secure because it is highly unlikely that two non-malicious values recursively produce colliding hashes through the BMT. This is an important primitive as a strong cryptographic hashing algorithm is one that randomly maps any data to a hash value with uniform probability, thereby making it impossible to compute a colliding hash value using brute force in a reasonable amount of time. When producing an 8-byte cryptographic hash, there is a probability p that two values will collide where p is $\frac{1}{8^8}$. In the baseline, when applying this to a BMT protecting 64GB, the resulting tree is ten levels (i.e., there are ten of these cryptographic hashes produced that protect this data). Thus, the probability that a particular path through the tree is produced for some particular data value is p^8 or $(\frac{1}{8^8})^{10}$. The likelihood that two independent values will have colliding hashes is $p^{10} \times p^{10} = p^{20}$, or 3.2×10^{-145}.

Cordelia modifies the hash size by producing a 7-byte cryptographic hash (as opposed to an 8-bytes) and has a theoretically minimum number of hashes being 2, regardless of memory size. As a result, the probability that two hash values will collide in Cordelia is p', where p' is $1/8^7$ and the probability that two paths through the tree may collide is as high as p'^2. Thus, the likelihood that two independent values will have a colliding hash is $p'^2 \times p'^2 = p'4$ or 5.17×10^{26}. While this is a significant reduction in the number of values to test for colliding hashes, the task of finding colliding hashes in Cordelia is still sufficiently difficult to uphold the secure memory properties. That is, if you could try 1000 data values per second (consistent with [224]), it would take approximately 4 years to come up with a fully colliding data path through the tree by brute force. Furthermore, this will be even increasingly difficult with more levels frozen from the Huffman part of the tree at the cost of some degree of adaptability.

3.5 Conclusion

Secure memory is a known bottleneck in memory intensive workloads. However, the performance impediment of secure memory should not necessitate its removal from secure hardware. In this chapter proposed Cordelia, an alternative approach to implement secure memory by reconsidering the construction of its underlying data structures. It showed that by considering the data structure as a means to optimize the secure memory protocol, Cordelia can improve performance of the system in a more scalable manner than by focusing on the metadata cache. Beyond this, reducing the authentication path length may have nice implications for emerging technologies, like non-volatile memories, which have limited device endurance.

With that said, Cordelia is not without its limitations. ① As detailed in Sec. 3.4, it requires less work to

create a collision in Cordelia than in a static alternative. ② Sec. 3.4.3 details the spatial overhead associated to maintain the auxiliary FGK structures with Cordelia. ③ Cordelia does not reconcile the issue identified in some prior work that updates to the tree shape cannot occur in parallel with authentication. As a result, Cordelia implements blocking behavior, which makes it difficult to reason about application performance. ④ Maintaining these structures in memory further motivates the need to reduce the spatial overhead of secure memory. One approach to achieve this is discussed in Chapter 4.

Chapter 4

Baobab Merkle Tree for Efficient Secure Memory

4.1 Introduction

As described in Chapter 1 and 2, secure memory has two key limitations against its practical deployment: (1) the memory authentication protocol requires additional work on memory fetch, which limits performance; and (2) secure memory metadata requires reserving a significant amount of in-memory space, which limits the amount of data accessible memory. While there has been significant work towards resolving (1) [52, 24, 116, 151, 318] (including the work proposed in Chapter 3), there has been far less work towards resolving (2) [256, 224, 298]. In fact, the proposed solution in Chapter 3, Cordelia, introduces *additional* metadata to further optimize secure memory for performance, and the approach of using additional metadata is consistent with several other works in the literature [317, 284, 276, 278, 274, 273, 275, 277, 183, 182, 316].

Resolving the spatial overhead of secure memory is an important problem. Every byte in a memory device reserved for metadata is a byte that cannot be used by the application. This is antithetical to the trends of modern computation, where applications use large quantities of data and have an *increased* demand for memory capacity. Such is the motivation for emerging disaggregated memory architectures (detailed in Chapter 6), which promise this increased capacity. In addition, constrained memory As a result, memory intensive jobs may be killed by the operating system or the application will need to rely on slower, long term storage to manage application data.

To alleviate this problem, this chapter proposes the *Baobab Merkle Tree*. The Baobab Merkle Tree takes advantage of the observation that many counters in memory have the same value. This phenomenon has been demonstrated in multiple of contexts [288, 192], and indicates that the space allocated to encryption counters may be inefficiently used. Given this observation, this work proposes an alternative protocol where encryption counter values are *memoized* in an on-chip table and the indices into the memoization table become the leaves of the integrity tree. As a result, the integrity tree memory overhead is shrunk to $2-4\times$ less compared to the BMT, as the index size is $2-4\times$ smaller than the counters. Furthermore, the Baobab Merkle Tree increases the likelihood of finding a metadata value in an on-chip metadata cache because a Baobab Merkle Tree node protects more data than its BMT equivalent.

This chapter presents the following contributions:

1. It propose the Baobab Merkle Tree, which memoizes encryption counters in an on-chip table, decreasing the spatial overhead of the integrity tree by $2-4\times$.
2. It defines a technique to memoize encryption counters on-chip.
3. It evaluates the Baobab Merkle Tree in gem5 [167], and discusses the trade-offs of its design.

4.2 Background

Seeing as Chapter 2 details the secure memory protocol in detail, the focus of this section is to overview the spatial overheads associated with secure memory metadata. In addition, this section examines the prior work that considers the overhead of these structures and the literature that examines how to reduce the spatial overhead of the metadata.

4.2.1 Spatial Overhead

As described in Chapter 2, secure memory metadata in a BMT[1] can be classified as being an encryption counter, a data MAC, or Merkle tree node. Each of these fields are explicitly designed to fit within a cache block (i.e., they are 64-byte aligned).

Recall that, in a BMT, each data word requires a MAC for authentication. The MAC is typically an HMAC (i.e., a cryptographic hash) of the data. Every 64-byte word of data has an 8-byte MAC associated with its state. That is, it requires 1 byte of capacity to store a MAC for every 8 bytes of data capacity. In order to maintain the MACs, a secure memory device must reserve $\frac{1}{8}$ or 12.5% of its capacity for the "MAC region" of metadata.

Furthermore, a BMT leverages counter-mode encryption (CME) [177] to encrypt its data state. In particular, the CME implementation use a split-counter design [298] (i.e., counters are composed by a major counter per page and a minor counter per word). A 64-byte block of counters consists of 64 seven-bit minor counters and a single 64-bit major counter. Each minor counter is associated with a 64-byte block of data, and the major counter is associated with each of the 64 data words in a page. This scheme implies that 8-bits of counter metadata (i.e., 7 bits in the minor counter and 1 bit from the major counter) is associated with each data word. That is, every byte of counter metadata is associated with 64-bytes of data capacity. Formally, this means that a secure memory device must reserve $\frac{1}{64}$ (1.56125%) of its capacity for the "encryption counter" metadata region.

Finally, a BMT constructs a Merkle tree on top of the encryption counters. The tree is 8-ary (i.e., each parent node has eight children), where node is composed of the concatenated 8-byte hashes of each of its children. The number of nodes in the tree is a function of the number leaves (i.e., encryption counters); this implies that the number of tree nodes is proportional to the data capacity of the memory device. More precisely, the number of nodes in the tree N in a memory with C encryption counters is defined as $N = \Sigma_{i=0}^{\infty} \lfloor (\frac{C}{8^i}) \rfloor$. The amount of space that a secure memory device needs to reserve is $N * 64$ bytes.

[1] Note, BMT refers to Bonsai Merkle Tree [222]. The Baobab Merkle Tree proposed in this chapter will be referred to with its full name.

Table 4.1: The amount of space required to store metadata in a secure memory for various capacities.

data capacity	MAC size	counter size	tree size	%age of data capacity
1GB	128MB	16MB	2.3MB	14.286%
256GB	32GB	4GB	585MB	14.286%
64TB	8TB	1TB	146GB	14.286%

Table 4.1 formalizes the amount of space that a memory device would need to reserve for various data capacities. Note, the percentage of space required for metadata *increases* (at a rate of less than .001%) as the capacity increases due to the increased number of nodes.

4.2.2 Related Work

The spatial overhead of the proposed BMT organization proposed in [222] is described in Sec. 4.2.1. This scheme is an improvement over the prior literature, which described building a Merkle tree directly over the data [247, 248]. Seeing as the encryption counters are more compact than the data itself, a BMT is built on top of fewer leaves. The resulting tree size is much smaller as a result. Further improvements in terms of spatial overhead require at least one of: ① reducing the number integrity tree nodes, ② increasing the density of encryption counters, and/or ③ reducing the MAC size.

To reduce the number of nodes in the Merkle tree, VAULT [256] proposes a scheme in which the arity of the integrity tree varies by level. The scheme targets trees of counters [81, 103, 99] and notes that nodes closer to the root will be updated more often than nodes towards the leaves because nodes nearer the root protect more data. Assuming data is accessed relatively uniformly, this means that it is safe to have fewer bits allocated per counter. Despite having fewer bits in these counters, the fact that they protect fewer data means that it remains unlikely that they overflow. The effect of having a higher arity is that fewer nodes are required in the integrity tree.

Morphable Counters [224] works from the observation that the space reserved for encryption counters is inefficiently utilized, even in a split-counter scheme. Much like in SYNERGY, this work makes the observation that the space allocated to metadata is under utilized. In this work, the number of counters in a cache block can dynamically shrink and grow as data is accessed at different frequencies. In particular, counters can be maintained in "normal" mode or in "compressed mode." As a result, a more compressed set of counters means that the resulting Merkle tree is built on top of fewer leaves, therefore the resulting tree requires less space as well. Note, when multiple potential metadata representations are possible, the secure memory protocol needs to be flexible to either case and some further metadata is required to distinguish between these cases.

SYNERGY [225] proposes a scheme in which MACs are co-designed with ECC bits. Seeing as ECC is also a metadata field, these bits also require space to be reserved from the application in the memory device. This work observes that ECC achieves two goals: error detection and error correction. A MAC also achieves error detection. Therefore, the work proposes using these MAC values to perform error detection and a small hint to inform correction. As a result, the spatial overhead of otherwise needing to maintain both ECC and MACs is significantly reduced. This scheme has led to several follow on works [72, 73, 173, 194] that similarly use MACs to help facilitate ECC.

In addition to increasing tree size, the VAULT protocol also implements, where possible, compressed

MACs [256]. Some MACs are compressible, so this protocol compresses them where possible. If MAC compression isn't possible, then the protocol describes potentially using a single MAC to protect multiple data. In this case, authenticating the data requires fetching all of the data that goes into that computation. Much like in Morphable Counters, this scheme requires a mechanism to delineate between which of the dynamic schemes are active for that protection.

4.3 Design

The Baobab Merkle Tree is a modification of the traditional BMT design that adds a single layer of indirection. The tree, instead of protecting each data block's counter, now protects the block's associated *index* into a *memoized counter table*. The table, containing all counter values, is stored within the on-chip memory controller. The table is divided into rows (i.e., *entries*), and each row contains a group of encryption counters (i.e., *cells*). Each data block is assigned to a fixed memoization table row, and its associated index (from the tree) indicates the column of its current counter value. The total number of cells in the memoization table is significantly smaller than the number of data blocks—the indices allow blocks to share counter values.

4.3.1 The Memoization Table

The memoization table describes a fixed size buffer. This buffer is composed of r memoization table *entries*, and each entry has c cells. The data stored in each cell reflects a counter value that can be used for counter-mode encryption. This chapter works from the premise that the memoization table is maintained on-chip. The reason for this is to ensure that encryption counter values fetched from the table are trusted without additional authentication. In doing so, there are limitations imposed on the maximum potential size of the table due to the on-chip area consumed by the table. If the on-chip area is constrained, this table can either be partially or totally moved to memory but the values would require further authentication. Designs that consider how to design an in-memory memoization table are beyond the scope of this chapter.

Unlike traditional secure memory designs, the proposed design does *not* implement a split-counter approach but instead uses a single block counter [298]. To reduce the likelihood of overflow and maximize utilization of space, each counter in the memoization table occupies $(64 - n)$ bits, which essentially resembles the traditional major counter in the split-counter design. When incrementing a counter value (described in Sec. 4.3.3), the Baobab system needs to consider the number of blocks currently using said counter value. Thus, the proposed design includes a reference counter to track the number of blocks actively using the encryption counter value. Only the $64 - n$ bit counter is used for encryption/decryption, not the reference counter.

The remaining n bits of the column values are used to keep track of the number of blocks currently using the counter. These bits represent a "sticky counter" [233], commonly used for reference counting. For example, suppose the scheme assumes a 60 bit encryption counter and 4 reference counter bits. The 4 bits are incremented every time a new block uses the counter value and it is decremented when a block changes to a new counter value. When the 4 bits reach their maximum value of 15 (i.e., `0xf`) the reference counter reaches the "non-decrement state." and can only be reset by finding all blocks pointing to it and re-encrypting them with a new counter value. The Baobab Tree takes this approach because it affords a cell to be pointed to exclusively over time, and helps maximize in-place incrementation, which is discussed in

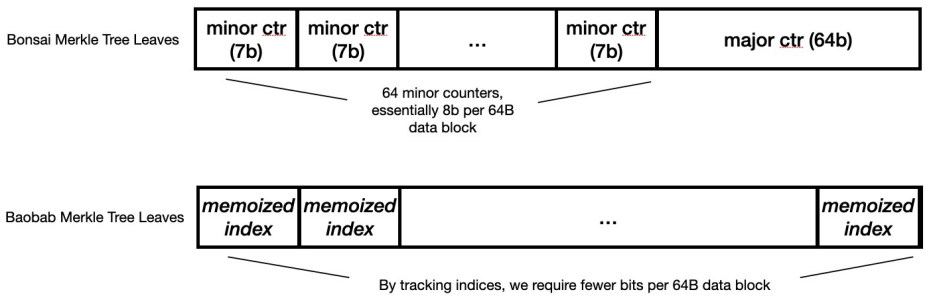


Figure 4.1: Layout of data in Baobab Merkle Tree leaves. Fewer bits are required for tracking the index of a cell in a memoization table entry than for major/minor counters.

more detail below.

4.3.2 Baobab Merkle Tree

The Baobab Merkle Tree is a tree of indices rather than a tree of counters. The leaves of the Baobab Merkle Tree are composed of n indices and each value is composed of $log_2 c$ bits, where c is the number of cells per memoization table entry. The physical address of the data determines where the cell index is stored, which is similar to how encryption counters are found in the traditional BMT design. Once the leaf node storing the index is accessed, the value stored determines the cell in the memoization table entry with the counter to be used for en/decryption.

4.3.3 Incrementing Counters

Incrementing a counter in the memoization table depends on its reference count and the state of the other counters within its entry. In particular, there are four types of increment scenarios in the memoization table: (1) in-place increment (2) next-cell increment, (3) free-cell increment, and (4) blocking increment. Fig. 4.2 demonstrates each case.

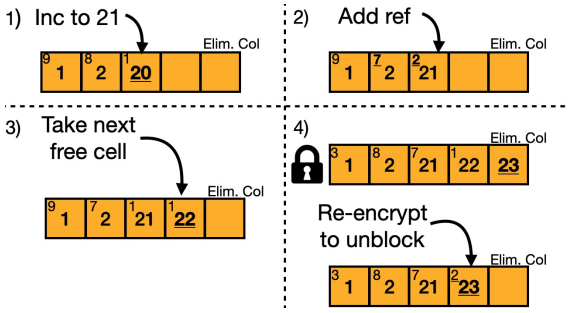


Figure 4.2: Incrementing counters using the elimination column.

In-place increment occurs when the block that requires incrementing the counter is the only block using that cell (Figure 4.2, scenario 1). If the current cell holds the largest counter value in the entry or if its counter value is at least two less than the next highest counter value (to avoid duplication of counter values), then it is safe to increment the current counter value in the column. The corresponding index in the Baobab Merkle Tree does not need to change. As such, no secure memory metadata access to main memory is required because leaves in the Baobab Merkle Tree refer to indices, which in this case do not change. This is a performance savings versus baseline BMT implementations.

Next-cell increment (scenario 2) occurs when the data is mapped to a cell with a reference counter greater than one and where there is another cell in the entry with a higher encryption counter. It also occurs when

a cell has a reference counter of one and another cell in the entry has an encryption counter one more than the current encryption counter to avoid duplication of counter values. In this case, the data block needs to now use the index of the next greatest encryption counter in the row. As such, this new index is stored in the Baobab Merkle Tree, whose state has changed requiring an update to the tree. In terms of memory operations, this case exhibits similar behavior to a standard write in a BMT. If the conditions for both in-place increment and next-cell increment exist, the memoization table chooses to use the next-cell behavior so as to ensure that the same counter value cannot occur twice within an entry. Most operations will either be in-place or next-cell incrementations.

Free-cell increment (scenario 3) occurs when data uses the cell with the highest encryption counter which is not held exclusively, but another cell in the entry has a reference count of zero (i.e., a free cell). In this case, the increment uses the free cell, filling it with the value of the incremented prior encryption counter.

Blocking increment (scenario 4) occurs when the data uses the cell with the highest encryption counter with a reference counter greater than one, and the entry has no free cells available to reuse. For this case, the system reserves the last column of the memoization table entry as the "elimination column." Its usage is depicted in Fig. 4.2. Suppose, after some time, the entry takes the state of the upper row in Fig. 4.2 scenario 4. In order to increment from 22 (i.e., next cell increment), the elimination column is filled. This locks further authentications to the entry to avoid conflicts. Then, in the lower row, unblocking is achieved by scanning for the least referenced cell in the entry and re-encrypting those data with the new encryption counter value created in the elimination column (i.e., re-encrypted with 23). The encryption counter from the elimination column then replaces the cell with the fewest references, and that data is re-encrypted with the new counter value. To find which data need to be re-encrypted, the scheme needs to perform a *reverse mapping* from counters to data to check which data points to that column and needs to be re-encrypted. To ensure that there is adequate hardware, all authentications that require this memoization row are blocked while re-encryption is happening. This is necessary so as to ensure that there are not two unique counters that both need the elimination cell in the same memoization table entry in parallel. In the worst case, each of the SNZI counters will be in the non-decrement state. When this occurs, upon doing the reverse mapping scan, the SNZI counters are updated with their correct value (which can reset them away from non-decrement state) and the truly least pointed to cell is determined.

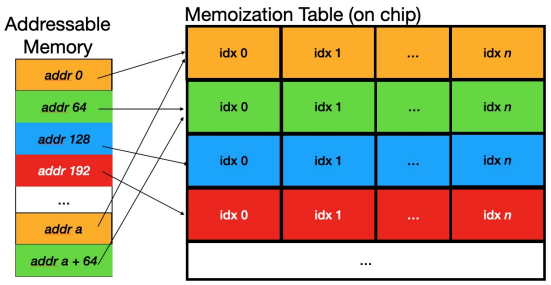


Figure 4.3: Memory assignment from address to memoization table row.

4.3.4 Assigning Blocks to Memoization Entries

The assignment of data to memoization table entries is an important feature of the Baobab Merkle Tree. To improve effectiveness, assignment of data to an entry works from a heuristic to increase the likelihood of in-place increment and decrease the likelihood of needing a blocking increment.

This scheme works from the observation that, like virtual memory, physical memory exhibits spatial locality (especially within a page). As such, contiguous data blocks (64 bytes) within a page should be

mapped to different memoization table entries. By doing so, the frequently used data within a page will have its counters increase monotonically in-place in different memoization table entries. If no physical locality is observed, blocks will need to increment counters at similar but slightly different rates, which will occupy more cells per entry. The scheme "stripes" the memory address in their mapping to memoization table entry, as per Figure 4.3. That is for each address $(e + i)/64$, where e is the number of memoization table entries and each data block is 64 bytes, that address will be mapped to entry with index i in the memoization table.

4.3.5 Security Implications

In order to uphold secure memory semantics, Bonsai MTs protect the integrity of encryption counters and use data MACs to ensure that data has not been corrupted [298]. The intuition is that only the uncorrupted encryption counter can produce the decryption key that decrypts the data to plain-text that matches the MAC. In the Baobab Merkle Tree, counters cannot be tampered as they are stored on-chip. Any attempts to tamper or replay the pointer will be detected by the integrity tree in the exact same way that the BMT would detect tampering or replaying of encryption counters in memory.

4.4 Evaluation

4.4.1 Methodology

The Baobab Merkle Tree is implemented as an extension to gem5 [167], a cycle-accurate full system simulator. The simulator is configured with four-cores where each core has private L1 and L2 caches, with a shared 8MB L3 cache. The integrity tree is 8-ary, and the "leaf" arity is n-ary (configuration dependent, but either 128-ary or 256-ary, described below). THe processor use a 32kB metadata cache and a 224kB memoization table. Each cell in the table is 8-bytes, with 58 bits belonging to the encryption counter and 6 bits acting as the sticky reference counter. There are two baseline approaches, one with a comparable metadata cache size to the Baobab Merkle Tree (i.e., 32kB metadata cache) and one with a comparable on-chip resource size (i.e., 256kB metadata cache). The simulations use SimPoint to determine the region of interest in each benchmark, and run 500 million instructions from this region of interest. In order to avoid inaccuracies in modeling due to cold-boot, the simulations begin by prefilling the memoization table state. The prefilled contents are collected from memory traces of each of the SPEC 2017 CPU benchmarks [41] run back-to-back while modeling what the table state would be offline from the simulation. The evaluation is performed using the SPEC 2017 CPU benchmarks and the Belgian street network workloads from the GAP benchmark suite [26].

4.4.2 Spatial Overhead

The spatial overhead of Merkle Trees in secure memory scales proportionally to the overall memory size. Table 4.2 shows the amount of reserved memory space required to store the integrity tree across configurations, showing both a *Baobab* and traditional *Bonsai* Merkle Tree. The fact that the scheme can protect and authenticate twice as much data per leaf in the Baobab Merkle Tree versus the Bonsai means that the Baobab Merkle Tree *requires half as much space* in memory as the Bonsai MT.

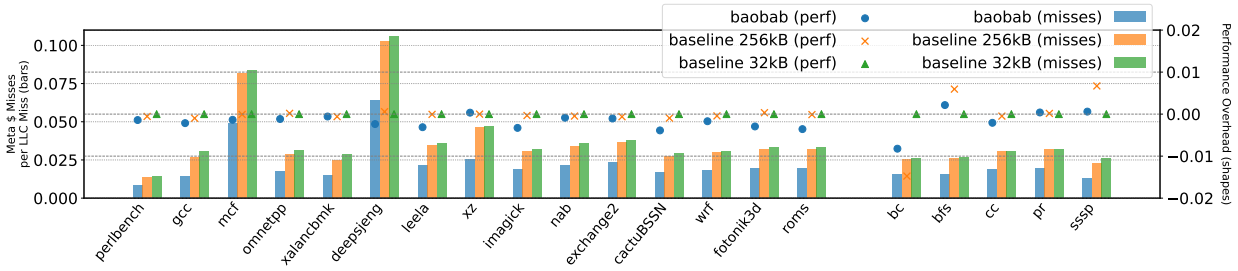


Figure 4.4: Evaluation across SPEC CPU 2017 and GAP benchmark suites. (Shapes) Execution overhead of the benchmark normalized to baseline secure memory protocol with 32kB metadata cache. (Bars) Metadata cache misses per LLC miss. In both metrics, lower is better.

The Baobab Merkle Tree size strictly depends on the number of cells within a memoization table entry. If, for example, 4 cells per entry are stored, then only 2 bits are required to track the index into the entry, and thus the Baobab Merkle Tree has a spatial reduction of $4X$ rather than $2X$ (256-ary versus 128-ary leaf level). However, this evaluation opts for 16 cells per entry in order to limit the number of blocking cases, requiring 4 bits to index into the memoization entry. Blocking cases can be done in parallel with accesses to different memoization table entries, so they do not impact performance, but they should still be avoided as much as possible to reduce the bandwidth requirement to service these requests.

The trade-off of reducing the number of cells per entry is that it reduces the number of possible "in-use" counter values per data. That is, with four cells per entry, all data assigned to that entry will be in one of four possible states at any moment in time. The drawback of having fewer possible "in-use" counter values is that it increases the likelihood of reaching a blocking case and reencryption may be more expensive because there are fewer alternative locations where this data to be stored.

4.4.3 Runtime Evaluation

This evaluation uses the SPEC 2017 CPU benchmarks [41] to evaluate the overhead of the Baobab Merkle Tree over a BMT. In general, the evaluation demonstrates negligible overhead of the Baobab Merkle Tree versus the baseline protocol with comparable metadata cache size and the comparable on-chip size. SPEC CPU is a general purpose benchmark suite, and as such provides an intuition of how the Baobab Merkle Tree would perform on average. In these cases, the Baobab Merkle Tree does not exhibit any performance overhead over the BMT. Figure 4.4 (shapes) shows that, the normalized performance of the Baobab Merkle Tree relative to the baseline BMT protocols (with a 32kB metadata cache) in terms of normalized number of cycles for the SPEC 2017 CPU benchmarks [41]. on average, the Baobab Merkle Tree implementation does not impact performance; it has an average performance benefit of less than two percent. That is, any

Table 4.2: Description of the spatial trade-offs in the Baobab Merkle Tree for varying memory sizes.

	Blks/Row	Baobab MT	Bonsai MT
1GB	66K	**9.5MB**	19MB
8GB	524K	**76MB**	153MB
256G	16M	**2.5GB**	5GB
1TB	67M	**9.5GB**	19GB
8TB	536M	**78.5GB**	157GB

differences in performance cannot be attributed to anything other than noise. As per [288], the latency to update a memoization table entry is 2ns, which is negligible relative to the memory access latency. For this reason, the overhead due to indirection incurred by the Baobab Merkle Tree is similarly negligible. While there is some additional information being tracked in the memoized data itself (i.e., the sticky reference counters), updating these values can be done at the same cycle and do not incur additional execution overheads. Furthermore, the evolution finds that the metadata cache hit rates are very high in the baseline approaches. In particular, the evaluation shows that only *perlbench* exhibited metadata cache hit rates below 97%. Even memory intensive benchmarks such as *mcf* and *lbm* exhibit strong re-use locality of security metadata. Given these factors, the Baobab Merkle Tree has no significant overhead relative to the baseline secure memory model, the impact of reduced metadata cache misses does not show its head in terms of overall performance. However, the evaluation also shows that the Baobab Merkle Tree does not have a negative impact on performance.

The Baobab Merkle Tree has a significant reduction in metadata cache misses relative to the Bonsai MT baseline, even though more on-chip space is used by the metadata cache. Figure 4.4 (bars) shows the number of overall metadata cache misses per last-level cache miss, comparing Baobab against the baseline secure memory systems with different metadata cache sizes. In every case, Baobab makes better use of the metadata cache capacity, resulting in a reduction in metadata cache misses.

Updates to the memory state do not necessarily result in updates to the underlying Baobab Merkle Tree state. As described in Sec. 4.3, the tree only needs to be updated when the index into the memoization table changes. In the BMT, every write to memory requires incrementing the minor counter protecting that data. Given this, the Baobab Merkle Tree can have a reduction of about 2% of updates to memory in 620.omnetpp_s benchmark. This is a function of the benchmark's hotness within a particular region of physical memory. Other benchmarks from the SPEC 2017 CPU suite do not exhibit the same levels of physical locality, and as a result do not have a reduction in the number of writes to physical memory.

4.5 Conclusion

This chapter presents the Baobab Merkle Tree, a secure memory protocol that describes a Merkle tree of indices into a memoization table rather than encryption counters. Seeing as indices require fewer bits than the counters themselves, the Baobab Merkle Tree reduces the spatial overhead of the integrity tree by $2-4\times$. Furthermore, by making data more compact, the Baobab Merkle Tree reduces metadata cache misses, which can be a promising approach for metadata cache dependent secure memory protocols. The Baobab Merkle Tree is a promising direction for future optimizations in both performance and spatial overheads of secure memory.

While the Baobab Merkle Tree sets out to mitigate the spatial overhead of the integrity tree structure, plenty of opportunities remain to reduce the spatial overhead of secure memory. The integrity tree is the *lowest* overhead component of the secure memory metadata fields. The biggest bottleneck continues to be the MAC structures. Although the work in [256] begins to address this issue, further reductions in MAC size begins to infringe on cryptographic primitives [92]. Appendix B overviews a novel proposed direction towards this end. At a high level, the idea is to consider the compression of *data* to improve the utilization of its associated metadata. If a memory device stores more data per word, then the utility of the associated metadata is increased.

The Baobab Merkle Tree is subject to further limitations. ① While the Boabab Merkle Tree requires less space than a normal BMT, the reverse mapping structure in memory is an additional metadata field that requires some storage capacity. ② Blocking authentications to a row in the memoization table can lead to similar performance degradation as the restructuring operation associated with Huffman reconstruction described in Chapter 3. As a result, it may be difficult for an application to reason about the longest latency memory access. ③ The memoization table on-chip proposed in the Baobab Merkle Tree requires more area than Intel SGX proposed in its memory controller extension. ④ As described above, there are several other metadata fields that dominate the spatial overhead of secure memory. In particular, storing MACs in memory requires significant space.

Chapter 5

A Midsummer Night's Tree: Efficient and High Performance Secure SCM

5.1 Problem Statement

Traditionally, "memory" has been synonymous with DRAM. Unfortunately, these devices put restrictions on the maximum capacity of a main memory given their limited density [202, 146]. Such is the motivation for emerging memory technologies. In particular, *non-volatile memories* (NVMs) describe a class of technologies that are a particularly attractive alternative as the transistor design is more amenable to scaling. Unlike DRAM, which relies on controlling discrete electrons for data storage, phase-change memories (PCM) uses analog current and thermal effects to manage data storage [146].

The notion of "volatility" refers to the storage of data in the device under a loss of power. A volatile memory device, like DRAM, requires power to perform row-refresh in order to retain its state. As a result, a loss of power implies a loss of data state in these technologies. Alternatively, NVMs retain their state through a power loss. As a result, NVMs provides the basis for storage-class memories (SCM) [197]. A SCM describes a non-volatile storage device that is accessible by a CPU through the same DIMM interface as a DRAM device. This naturally affords the design and deployment of server-scale applications that use memory as its long-term storage interface (as opposed to an external device accessed via I/O) [114, 134].

Programmers using computing platforms with SCM must account for the *crash consistency* of the data. That is, because application data in memory will be *persistent* through a crash, it is possible that merely restoring the device and resuming execution will lead to incorrect behavior [204]. On-chip cache state is inherently *volatile*. For instance, the correct and coherent version of some data may reside in the cache state on a loss of power, so the application may have an incomplete view of the program state on system restoration. Alternatively, the cache hierarchy may evict data from its state to be written back to memory in an arbitrary manner with respect to program order. In both cases, correct execution is achieved in practice by offloading the task of persistence to the programmer to explicitly implement [20, 84, 185].

To implement secure memory for an SCM, the memory controller must achieve a similar end as the programmer. If the secure memory protocol uses a metadata cache, which Chapter 2 indicates a is necessary prerequisite to achieve practical performance, then some metadata state is subject to inconsistency upon

system restoration. However, the potential crash inconsistency of metadata is critical for correct execution. Incorrect crash consistency can lead to incorrect detections of corruption thereby incorrectly rendering a system state maliciously tampered. As a result, the correct crash consistency of secure memory metadata is an issue of paramount importance.

5.2 Introduction

Traditional systems with volatile memory technology suffer from active and passive physical attacks where data in memory can be selectively targeted and corrupted [191, 213, 168, 124]. Protecting against these vulnerabilities must be done in hardware, and has been thoroughly investigated over the past two decades [53, 133, 292]. However, since volatile memory systems lose their state when power is disconnected, prior work did not have to address the data remanence problem [239]. Storage class memory (SCM) systems use non-volatile technologies as main memory, so data will remain intact even after power is disconnected. Such systems are natural candidates for large memory applications, where there is a lot of data that may be queried and stored. Disk storage is not viable for these cases due to its latency, and volatile memories are not viable due to the lack of persistent data storage. Moreover, SCM systems face new physical attack risks due to their application, providing attackers with extended opportunities for splicing, spoofing, and replay attacks. Thus, SCM requires a solution for protecting data in memory, such as secure memory guarantees and protections.

As described in Sec. 5.1, the key issue to a secure memory for SCM is to ensure the crash consistency of its metadata. To achieve this, the secure memory protocol *could* instead persist all values that are written to the metadata cache directly to memory. In doing so, the caching policy can be referred to as a write-through cache, as opposed to its typical writeback nature. This scheme, termed *strict metadata persistence*, is crash consistent because each of p_t, C_t, H_t, and T_t are persisted directly and atomically, so all values in memory are in a crash consistent state at all times. However, this scheme is not realistic, as it can lead to steep performance overheads (up to 25×) at runtime.

An alternative approach, dubbed *leaf metadata persistence* [301], addresses the performance issue by taking a *lazy* approach to crash consistency. That is, only p_t, H_t, and C_t are persisted directly at runtime. The tree nodes T_t are written to the volatile metadata cache and only written back to memory on eviction (i.e., they are not written-through directly). After a crash, at system recovery, each of the inner nodes of the integrity tree are recomputed from the hashes of its leaves (i.e., the counters). If the computed tree root matches the stored tree root, then the system can be safely rebooted. However, this recovery procedure is pessimistic because all inner nodes of the tree are assumed to be stale/untrusted, and recovery will be worse as memory capacities continue to grow beyond the scale of current SCM devices. These two extreme baselines describe an inherent trade-off between runtime *performance overhead* and *recovery time*. That is, performance overhead is reduced as crash consistency models become lazier, but at the cost of increasingly unreasonable recovery times.

The current state-of-the-art navigates this trade-off space, and can largely be categorized as being either a *static* [24, 231, 301] or *dynamic* [9, 284, 214, 86, 317, 8, 6, 50, 152] negotiation of performance overhead due to crash consistency and recovery time. Static approaches work well to strictly reduce the overhead due to maintaining the crash consistency of secure memory metadata [301], reducing the work required at recovery time [24], or partitioning hybrid untrusted device semantics [231], but these approaches miss out on potential

performance benefits by treating all addresses the same. Dynamic approaches, on the other hand, explicitly track application behavior to ensure that "hot regions" of memory benefit from having shorter paths through the integrity tree to persist values [214, 86], maintain an auxiliary "fast tree" in which frequently accessed values can be directed to a more relaxed crash consistency protocol [284, 9], or add auxiliary structures to further protect the metadata cache [317, 8, 6, 50, 152]. A limitation of these approaches, however, is that hot region tracking in hardware is difficult, and is dependent on application behaviors. Furthermore, these approaches tend to come at the cost of *hardware complexity*, a third component in this trade-off space that hasn't been emphasized to the same extent as performance or recovery.

Considering the cost of hardware complexity is important in terms of the performance scalability and security of secure memory. If the space for these devices occupies too much trusted space (i.e., on-chip), there will be less space for other devices like the last level cache, thereby causing more fetches to go to untrusted SCM and further binding application performance to the performance of secure memory.

This chapter proposes *A Midsummer Night's Tree* (AMNT) [1], a "tree within a tree" metadata persistence protocol that provides integrity-protected SCM with a low runtime overhead and a bounded recovery mechanism. AMNT's design goals are to achieve a crash recovery scheme with low runtime overheads, bounded recovery times, and maintaining limited area overheads both on-chip and in memory. AMNT achieves these goals by implementing a *hybrid metadata persistence protocol* that is *adaptive* to workload characteristics at runtime.

AMNT works from the insight that certain "hot" regions of physical memory may be accessed with more regularity, whereas an application may never access other regions. AMNT leverages this insight by implementing a hot-region tracking mechanism in which a small region in memory gets to benefit from a lazy metadata persistence scheme. As a result, only a small and bounded amount of memory will be stale/untrusted at the time of a crash, and the amount of metadata to recover is similarly small. In addition, AMNT gives a system administrator the ability to dictate the tolerable recovery time after a crash by selecting, in BIOS, the maximum stale data size (defined by the level at which the subtree root is placed). This chapter demonstrates that this insight holds true for several applications with varying characteristics. For adversarial cases, AMNT turns to software to modify behavior at the application layer to better take advantage of more tightly bounded physical regions of memory, which minimizes AMNT's physical area overhead.

In summary, this chapter makes the following contributions:

- It presents AMNT, a dynamic hybrid metadata persistence scheme for secure SCMs that performs hot region tracking to adapt to in-memory behaviors at runtime.
- It introduces AMNT++, an optional hardware-software co-design physical page allocator that acts as an addition to AMNT in order to improve the likelihood of an in-use page to be tracked in the hot region.
- It shows how AMNT uses 96 bytes of volatile on-chip space and 64 bytes of non-volatile on-chip space, which is agnostic to memory and metadata cache size.

[1] In William Shakespeare's play *A Midsummer Night's Dream*, the Mechanicals perform a play called "The Most Lamentable Comedy and Most Cruel Death of Pyramus and Thisbe," which is known as a "play within a play."

- It demonstrates how a system administrator can bound the recovery time using this proposed approach to achieve desired performance goals.

5.3 Background

Seeing as the secure memory protocol is overviewed in detail in Chapter 2, the goal of this section is to overview the challenges of implementing secure memory in an SCM device and provide an intuition into some of the related work in the area.

5.3.1 Secure SCM

Storage class memories (SCMs) describe a system based on emerging technologies in which main memory is *non-volatile*. These devices are appealing in that they promise near-DRAM latency with the persistence properties of long-term storage devices. As a result, they are a natural candidate for data storage applications where performance is bound by disk accesses. These types of applications use main memory to enforce persistent semantics. Unlike traditional DRAM-based memories, data in SCM will *persist* through a crash (i.e., its state is retained without power). Thus, SCM systems face the challenge of ensuring the crash consistency of secure memory metadata. When values are updated in on-chip volatile caches, their values become stale in main memory due to its non-volatility. Barring additional action, in the event of a crash, the up-to-date values in cache will be lost, leaving behind stale values in non-volatile memory and rendering the program unrecoverable on reboot forgoing the benefits of SCMs.

Various software libraries (e.g. [47, 126, 210, 280, 60]) and hardware extensions (e.g. [313, 196]) provide the programmer sufficient control to avoid data inconsistency problems and ensure application data is crash consistent after a power failure. Like application data, security metadata (i.e., HMACs, counters, and the BMT) must also be crash consistent to ensure that data can be authenticated after a crash. Unlike application data, secure memory metadata is not accessible to the application, requiring the on-chip memory controller to enforce its persistence. An implication of this phenomenon is that the root of the BMT must always reflect the state of data in main memory, and updates to the state and the root must be atomic. Furthermore, the root of the BMT must be stored in a non-volatile on-chip register in order to be trusted through a crash.

In general, strategies for ensuring metadata is usable after a crash fall on two extremes. A *strict metadata persistence* strategy ensures that all BMT values are consistent with the state of the data in memory at all times – on a crash, all metadata is guaranteed to be stored in a non-volatile device and can be used for integrity verification. However, this method exhibits high runtime overhead. On a data write, each node in the ancestral path of the BMT must be updated in the on-chip metadata cache and written-through to main memory. While this technique is expensive at runtime, recovery is trivial, as all metadata is immediately available on restart.

By contrast, a *leaf metadata persistence strategy*, while improving runtime performance, significantly increases recovery time. In this strategy, only the BMT leaf (i.e., counter) and root updates are done atomically with a data write. The rest of the security metadata is updated lazily on a writeback from the metadata cache. On system failure, all inner BMT nodes must be assumed to be stale in SCM, and must be recomputed. In order to recompute the BMT nodes, BMT leaves must be fetched and their hashes computed. Inner-BMT nodes are composed of the keyed hashes of their children, which makes the computation of a

node in the BMT dependent on the fetch of each of its children. The data dependent nature of BMTs limits the number of productive parallel memory fetches to BMT sibling nodes, and implies that a large number of bursts (proportional to memory size) must be performed in order to recompute all of the inner BMT nodes. As such, recomputation can last billions of cycles and spans all of secure memory metadata, which, for SCM, may run into the terabytes. Once recomputed, the BMT hashes are compared to the root, which is stored securely and persistently on-chip.

5.3.2 Prior Literature

Osiris [301] further relaxes the leaf metadata persistence protocol by introducing a "stop-loss" persistent metadata cache for BMT leaves. The protocol persists leaves after every n data updates to ensure that they can never be more stale than the stop-loss frequency. As a result of the persistence relaxation, full BMT recovery in Osiris is slower than in leaf metadata persistence. Combining persistence techniques is not novel. Prior art has proposed partitioning the persistence policy of some metadata statically based on its tree level [24] or based on its residence in volatile or non-volatile memory [231]. Prior art suggested statically partitioning the persistence policy of metadata based on its tree level [24] or its location in volatile or non-volatile memory [231]. However, there is no prior work that proposes a dynamic persistence scheme. In order to recover leaves on a crash, each counter is checked against a MAC stored in the ECC bits of the data, and the counter is incremented if the comparison fails.

Persist Level Parallelism [88] focuses on fast integrity tree updates and explores the benefits of having parallel updates of the BMT under strict conditions that guarantee correct crash recoverability. However, these works are not *dynamic* to changing application behavior and as a result do not see the benefits in terms of recovery time in the case of Osiris nor normal case runtime overhead in the case of Persist Level Parallelism.

Anubis [317], much like AMNT, provides low run-time overhead and fast recovery. However, Anubis takes a fundamentally different approach to crash consistency, and enforcing this protocol has implications on runtime behaviors and overheads. Anubis tracks the metadata address currently residing in the volatile state (i.e., in the metadata cache) in persistent memory. This region of memory, dubbed a "shadow table", essentially creates a log of all potentially stale metadata values in the persistent state at the time of a crash. Thus, any update to the cache state (misses, evictions, and writebacks) require updating the state of the in-memory shadow table. However, because this table resides in untrusted memory, it too needs to be protected by an auxiliary "shadow Merkle tree" to preserve its integrity. Thus, updates to the shadow table also result in updates to the auxiliary "shadow Merkle tree." While this protocol results in low runtime overhead in most cases, it makes the case of a metadata cache miss more expensive. Anubis works from the observation that updating this table is infrequent because the metadata cache tends to exhibit good locality. We further note in our evaluation that updating the shadow table needs to be atomic with updating the tree state, and there may be multiple shadow table updates on a single authentication (due to multiple misses in the metadata cache). Furthermore, it requires caching the entire shadow Merkle Tree on-chip to avoid even more memory persists per data access. By contrast, the AMNT protocol trades off tracking of the stale nodes for minimal area overheads while still bounding the recovery time. Furthermore, AMNT is not bound by the hit rate of the metadata cache. Instead, it is dependent on a hot region tracking mechanism, in which the complexity is offloaded to software.

Bonsai Merkle Forest (BMF) [86] is a protocol designed to dynamically reduce the leaf-to-root write path for frequently accessed nodes. To do so, BMF extends the persistent register used to store the BMT root into a non-volatile metadata cache to store several roots of frequently accessed values (dubbed the "persistent root set"). BMF determines which BMT nodes qualify to be stored in the persistent root set, and tracks accesses frequency counters for the cached blocks in the non-volatile metadata cache. On a pre-determined interval, BMF uses these access frequency counters to "prune" a frequently accessed root into its most frequently accessed children or "merge" colder roots into their parent node. This mechanism ensures that all nodes in the BMT are covered by a persistent root, which is an important correctness property for this approach. However, this property also implies that it is infeasible to perform a hybrid metadata persistence strategy. As a result, it suffers from the limitations of whichever crash consistency policy it implements. More frequently accessed subtrees are tracked by persistent on-chip subroots to ensure that nodes in these paths do not need to persist updates from the leaves to the true root of the tree. These changes to the set of cached nodes changes dynamically, and incrementally across intervals. For BMF to work, it requires that all leaves are protected by a subtree. However, their work focuses on improving performance of BMT root updates by expanding the persistent domain, and recovery time is not a primary concern. In order for BMF to be recoverable in a reasonable amount of time, its underlying protocol must implement strict persistence. Like BMF, AMNT considers the frequency of accessed nodes for persistent state but leverages this frequency to improve runtime performance and bound the recovery time to a well defined limit. Unlike BMF, AMNT does not assume full leaf coverage of a fast subtree, so it can improve runtime overhead without jeopardizing fast recovery. AMNT's protocol does not entail incremental changes to track hot nodes, and as a result exhibits better performance than BMF. Furthermore, AMNT does not require buffers of non-volatile memories on-chip, to cache the large number of roots required for full leaf coverage as BMF does.

Some prior works consider multiple persistence protocols in secure memory to benefit performance [231, 24]. For example, Triad-NVM describes a protocol in which entire levels of the tree conform to a particular persistence protocol [24]. Such a behavior is similar to leaf persistence, where leaves and some number of ancestral levels of the leaves are written-through to main memory. AMNT similarly implements multiple persistence strategies, but determining which strategy to use is done dynamically based on application behavior. OMT describes a protocol for hybrid embedded volatile and non-volatile memories, that leverages a single integrity tree with different strategies for data in different devices [231]. AMNT abstracts well to a hybrid SCM-DRAM machine as it does not require significant protocol or hardware changes. AMNT protects SCM, and a traditional BMT protects DRAM. This solution only requires an additional (volatile) register for the BMT and knowledge at the memory controller of the SCM/DRAM physical address partition. Furthermore, AMNT dynamically leverages leaf metadata persistence in the most frequently accessed SCM region.

5.4 Threat Model

This work assumes the threat model described in Sec. 2.2. Within this, it is worth noting that this threat model makes no assumptions about the software running on the underlying hardware platform. Therefore, a modification to the operating system does not provide an attacker any *additional* mechanism to exploit the physical vulnerabilities of a memory device. Other attacks beyond the scope of this work's threat model may be introduced, but defending against these are an orthogonal problem.

5.5 A Midsummer Night's Tree

The proposed solution, *A Midsummer Night's Tree* (AMNT), is a secure SCM protocol that balances a reasonable runtime overhead with controllable recovery times and minimal hardware overhead. AMNT achieves its goals by using dynamic hybrid metadata persistence strategies within the same BMT [222]. Applications tend to exhibit spatial locaity across phyiscal addresses, which leaves an opportunity to create a secure SCM system with both low runtime overhead as well as low hardware overhead. This section also proposes an optional mechanism to further optimize AMNT, called AMNT++, which adds a lightweight modification in the physical page allocator to further improve the design's performance.

5.5.1 "A Tree Within a Tree"

This work assumes that a small number of contiguous addresses in physical memory are frequently accessed (i.e., "hot"). Given this assumption, this section proposes AMNT, which protects a small region of physical memory with a fast persistence protocol while most addresses are persisted strictly to keep the work required at recovery time low. AMNT is a dynamic metadata persistence protocol that tracks hot regions of physical memory within a *subtree* of the underlying BMT in order to adapt to changing in-memory hotness at runtime. The subtree implements a leaf persistence strategy, where tree nodes are assumed to be stale at the time of a system failure (blue nodes in Figure 5.1). The rest of the BMT implements strict persistence to minimize recovery time after a crash (red nodes in Figure 5.1). Given that a small contiguous region of addresses are frequently accessed, AMNT makes updating them fast. Implementing strict persistence outside of the subtree, while slow at runtime, will not occur often, minimizing the impact on overall performance and reducing the work required at recovery time. Note, AMNT works *because* the number of leaves covered by the subtree is small. If AMNT were to implement a large number of subtrees, and a large number of leaves in the BMT were protected by the leaf metadata persistence policy, then the amount of time to recover after a crash would devolve towards leaf metadata persistence. By splitting BMT nodes into multiple persistence models, the recomputation required at recovery is a function of the subtree size.

Implementing the AMNT protocol involves splitting the BMT into the main tree with strict metadata persistence (slow runtime, fast recovery) and a subtree with leaf metadata persistence (slow recovery, fast runtime). The subtree root, situated at an internal BMT node, is placed in an on-chip non-volatile register; its descendants are expected to contain frequently-accessed data. The fast subtree register allows for data authentications to quickly determine their persistence protocol. This approach makes data updates within the subtree much faster—the associated tree node writes only need to be updated in the metadata cache. In contrast, if a data update occurs outside of the subtree, it will need to wait for all BMT nodes on the ancestral path to be written-through to persistent memory.

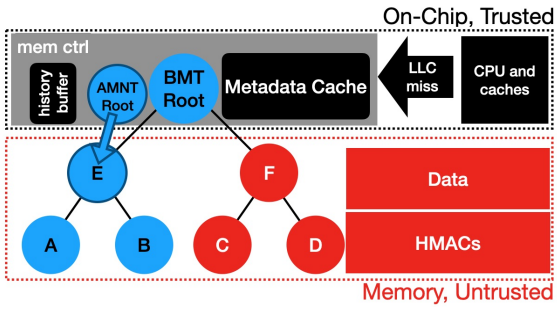


Figure 5.1: A Midsummer Night's Tree. Red nodes implement strict persistence. Blue nodes implement leaf persistence.

Given the metadata persistence strategy, all values outside the subtree root in the BMT are not stale at the time of a crash. To recover the BMT, AMNT only needs to recompute nodes inside the subtree; recovery time depends on the subtree size determined by the subtree root level. System administrators can control recovery time by configuring the subtree root level in the BIOS, and this work offers insight into the trade-off between recovery time and runtime overhead in Section 5.7.

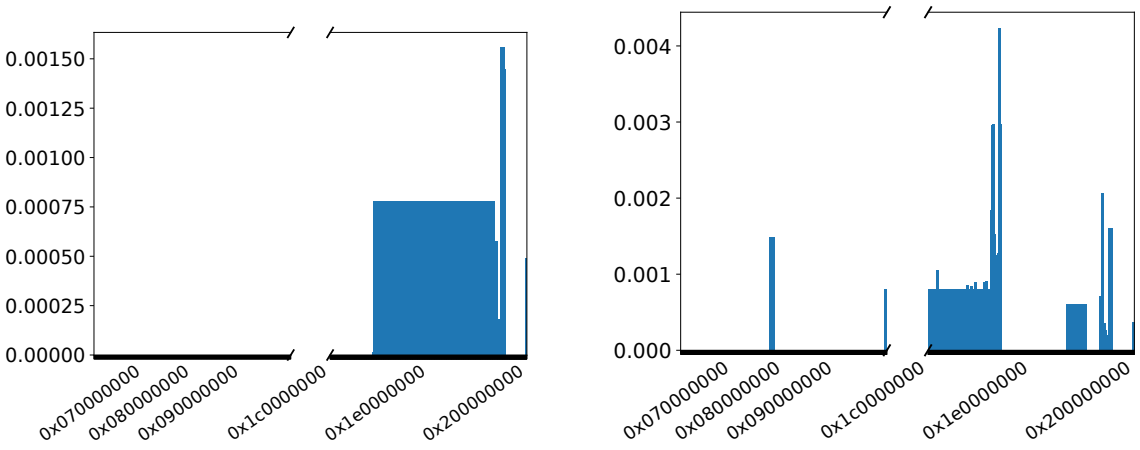
5.5.2 Hot Region Tracking

The AMNT protocol assumes the subtree root resides at a particular level of the BMT configured in the BIOS. Any node can become the subtree root at this level depending on application behavior in memory (i.e., the subtree root can move horizontally in the tree). Each node at this level protects a contiguous physical address space, termed the *subtree region*. In order to efficiently determine the most frequently accessed subtree region, AMNT makes use of a lightweight history buffer.

The history buffer has n entries and tracks the n most recent memory writes. Each entry has a subtree index (identified by the index of the node within the subtree level) and a $log_2 n$ counter. On a data write, the subtree index of the corresponding data address is updated by scanning the history buffer for that index and incrementing the counter. If the node becomes the most frequently accessed, swapping the node with the head element ensures the head of the buffer always refers to the most frequently used subtree region (the largest counter). While updating the history buffer can be done in parallel with the initial counter fetch (all authentications need to fetch the encryption counter), the history buffer is *not* fully sorted to minimize complexity. In this approach, the head element is guaranteed to be the maximum. After n data updates to memory (64 by default), the head of the buffer is selected as the new subtree root. After the next subtree root is established, the counters in the buffer get zeroed out and the tracking starts again.

When transitioning from subtree T to T', all inner integrity nodes of T must be persisted before T' can implement the leaf persistence protocol in order to preserve the crash consistency and security guarantees. Note that the only ancestral paths from subtree T that need to be written to memory are those originating from modified (dirty) data. AMNT can quickly determine which nodes need to be updated in memory by scanning the dirty bits in the metadata cache. Only nodes in the metadata cache that fall within the subtree will have their dirty bits set as all other metadata blocks are written-through to memory. The path from T to the root must always be persisted on movement.

The history buffer is a lightweight method to track the most frequently used regions of memory to select the best subtree root. Each entry in the history buffer requires at most $log_2 n$ bits for the region's index and an additional $log_2 n$ bits for the counter, resulting in $n * (2log_2 n)$ additional bits. For a subtree at level 3 (64 possible subtree regions), the additional number of bits is 768, requiring an additional 96 bytes of on-chip area. Thus, in practice operations such as scanning the history buffer to increment the frequency counter associated with a subtree region are inexpensive (two cache accesses) relative to memory access latency. Once found, the logic to update the buffer is a simple add and comparator that updates the head of the buffer based on if the target counter is larger than the head's counter. In the event of a tie, the current subtree root stays at the head of the buffer to avoid a subtree movement. Given that the updating the history buffer and transitioning the subtree are not critical to the authentication of data, they can occur out of the critical path of data authentication.



(a) Single program behavior (*lbm*).

(b) Multiprogram behavior (*perlbench* and *lbm*).

Figure 5.2: Memory accesses per address in single program and multiprogram workloads.

5.6 AMNT++

AMNT is predicated on the assumption that a small, contiguous address range in physical memory is protected by the fast subtree. However, while this assumption may hold true for a single application, it may not in a realistic scenario (i.e., a multiprogram environment). Fig. 5.2a shows the number of memory accesses per physical address in the *lbm* benchmark from the SPEC CPU benchmark suite, whereas Figure 5.2b shows the number of memory accesses per physical address when running two programs (*perlbench* and *lbm*) in parallel. As evidenced by the figure, multicore systems that run multiple applications prove the assumption driving AMNT may not exist in practice. To address this limitation, one could consider a protocol that has "per-core subtrees" to track hotness, but such a solution would result in complex and large hardware requirements for devices with hundreds of cores. Instead, AMNT proposes a *hardware-software co-design* called *AMNT++* to keep hardware complexity low and modify application behavior from the operating system's memory management unit to bias allocation of physical pages towards highly available subtrees in order to maximize potential subtree locality.

To increase the effectiveness of AMNT, all applications should ideally work in the same subtree region whenever possible to increase subtree locality. In order to further optimize AMNT, an optional *hardware-software co-design* mechanism to improve hit rates in the fast subtree may be deployed. Consolidating frequent memory accesses into a single subtree can be an important performance optimization, which is maximized through lightweight modifications to the physical page allocator in the operating system (OS). Lower subtree hit rates imply that AMNT will write-through more nodes of the underlying BMT. To help optimize subtree hit rates, AMNT++ utilizes the lightweight modified physical page allocator to better take advantage of the underlying hardware. Furthermore, modifying the OS is a minor modification relative to designing and adopting new hardware. As such, seeing as AMNT (or any other secure memory implementation) requires acquiring new hardware, the use of the OS modification is no more or less likely than the use of this novel hardware.

In Linux, allocating physical memory is a distinct procedure from allocating memory at the application level. Theoretically, cross-page locality may be unlikely given that physical pages will be allocated according

to a *binary buddy allocation* scheme and where "random" pages are reclaimed by the OS over time. This allocation makes it difficult to reason about where two virtual pages are in physical memory relative to each other.

In order to further increase in-memory physical locality, this approach modifies the buddy allocation from the Linux operating system [163]. This modified OS achieves this locality by reordering the free area to have the chunks within the most common free subtree region at the head of the linked list. Physical pages are allocated from a data structure called `free_areas` (i.e., an array of linked lists), where each linked list is composed of "chunks" of physical memory. The size of each chunk depends on the index of the linked list in the array (e.g., chunks in a linked list at index 0 of the `free_area` are 2^0 pages; chunks at index 1 are 2^1 pages). When an allocation request for a single page is received, the physical page allocator fetches the first item from the `free_area` linked list at index 0, and returns it to the application. When the linked list at index i is empty, and the OS needs to allocate a physical page, it will attempt to find a chunk at index $i+1$. If it finds a chunk at $i+1$, it splits that chunk into two chunks of size 2^i pages and returns one to be allocated while adding the other one to the linked list at index i.

In the modified buddy allocator, the linked list structure is modified to prioritize chunks that are physically close to one another and placing these at the head of the linked list. As physical memory is reclaimed by the OS, it attempts to add chunks to the linked list at the appropriate index of the `free_area` depending on the chunk size. In the AMNT++ modification, the linked list is reordered to place chunks within the subtree region at the head of the linked list. This approach makes each individual allocation as fast as the standard physical allocator by taking the restructuring of the linked list out of the critical path of a physical allocation.

The AMNT++ restructuring function is called during the OS physical page reclamation procedure, leaving it out of the critical path of a page allocation. The restructuring function first scans each linked list to count how many chunks fall under each subtree region. When the OS finishes scanning the list, it selects the region with the greatest number of chunks (the subtree region with the most free chunks) and then moves all the chunks for that region to the front of a temporary biased linked list (not in the `free_area` struct). Once the OS is done with the restructuring, the OS replaces the linked list with the new biased version. While the restructuring of the list is expensive, its infrequent occurrence largely amortizes its cost. The OS tracks the number of chunks that fall within that region; when the number of allocations matches the number of existing chunks in that region, the restructuring procedure is triggered (during reclamation).

The design decision to bias the allocator towards physical pages at the granularity of subtree regions is intentional. The AMNT++ protocol biases the `free_list` towards pages in the subtree region, so that the OS does not have to perform these exhaustive searches when allocating pages. Given that the locality of interest is in terms of the integrity tree nodes coverage (e.g. at level 3 the coverage is 128MB for an 8GB memory and scales with increased memory size), achieving this locality is a reasonable task, even in a fragmented system. As a result, the benefit of AMNT++ is high without being overly intrusive on the execution of the OS and application. For example, the OS should not impede the applications from allocating pages when pages in the fast subtree are free, but not at the head of the `free_list`.

Security Configuration	
BMT	8-ary integrity nodes 64-ary counters
Metadata Cache	64kB, 2-cycle latency
AMNT	64 writes per interval Subtree Level: 3, 768 bit history buffer, 128 bit dirty path bitmap
DDR-based PCM Configuration	
Capacity	8GB PCM
Latency	305ns read [127], 391ns write [113]

Table 5.1: AMNT Configuration for Evaluation.

5.7 Evaluation

This section first evaluates AMNT on the PARSEC benchmark suite [32] version 3.0 with the simlarge inputs in gem5 [33], a cycle accurate processor simulator. The processor configuration includes a single core with a 32kB data L1 cache, a 48kB instruction L1 cache and 1MB L2 cache. The simulator is configured with a processor that uses intentionally small on-chip caches to stress the memory system and show the overheads of the secure memory hardware. The memory system configuration is shown in Table 5.1. The secure memory hardware includes a 64kB metadata cache with an 8-level BMT to remain consistent with Intel SGX's configuration [109].

This evaluation uses PARSEC as the primary means of evaluation due to its diversity of workloads and labeling of the beginning and end of the region of interest. The latter is important—determining the region of interest in benchmark suites without labeled regions of interest is typically done by running a profiling tool, like SimPoint [205]. These tools determine the region of interest based on microarchitectural characteristics of the workload, and the application is run from this point for a set number of instructions. However, this work cannot compare AMNT and AMNT++ using this methodology as the modified OS results in a different number of instructions, requiring different points in the program to execute the same region of the application.

The evaluation consistently compares the proposed approaches (AMNT and AMNT++) against the **leaf** metadata policy and the **strict** persistence policy. In addition, it also compares the proposed approaches against various protocols proposed in the literature, **anubis** and **bmf**, which are implemented based on their design descriptions [317, 86]. Results labeled as **amnt** show the proposed protocol without the modified operating system, and **amnt++** show AMNT with the modified operating system.

5.7.1 Single Program Analysis

On average, AMNT has a 16% performance overhead relative to the volatile secure memory scheme and AMNT++ has a 10% performance overhead. Fig. 5.3 shows the cycles for each configuration normalized to the volatile secure memory results. Leaf and strict persistence have 8% and 2.39× performance overhead respectively. AMNT effectively negotiates the trade-off between leaf and strict persistence, achieving its design goal of having the near-leaf performance overhead.

On the other hand, Anubis [317] largely benefits from leaf persistence, but incurs a slow-path case on a metadata cache miss. For workloads that have bad metadata cache efficacy (i.e., *canneal*), Anubis results in large performance overhead, 2.4× compared to the volatile secure memory system. That is, Anubis is

predicated on the assumption that the metadata cache exhibits strong locality, but in *canneal* has 30.4% metadata cache hit rate. In contrast, AMNT's performance is directly dependent on the application's spatial locality, as opposed to cache efficacy, and as a result is able to lower *canneal*'s overhead down to less than 0.1%.

5.7.2 Multiprogram Analysis

As described in Section 5.6, running a single program may not fully stress the underlying protocol in AMNT as the single program's address space will be the only one exhibiting locality in memory. Thus, this section uses multiprogram workloads to approximate real-world behaviors in which the memory system is subject to the interference due to the interaction of multiple processes. In order to perform such an evaluation, the evaluation configures the environment with combinations of programs with temporarily similar regions of interest from the PARSEC benchmark suite. The multiprogram evaluation methodology is consistent with prior work [272]. To choose the pairs of multiprogram workloads, the benchmarks whose region of interest appeared at the most similar times are selected to ensure that the regions of interest of each benchmark is evaluated in parallel. These workloads are: *bodytrack* and *fluidanimate*, *swaptions* and *streamcluster*, and *x264* and *freqmine*. To ensure that the evaluation always covers both benchmarks' regions of interest, the analysis starts measuring when the second benchmark reaches the beginning of its region of interest, and stop the simulation when the first benchmark reaches the end of its region of interest. The presented results come from running both region of interests in parallel. The simulator configuration for the multiprogram analysis includes two cores, each with a private 32kB data L1 cache, 48kB instruction L1 cache, and 128kB L2 cache. Both cores share a 1MB L3 cache.

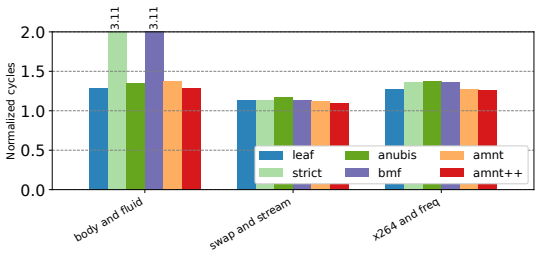


Figure 5.4: Normalized cycles in multiprogram PARSEC workloads.

AMNT++ is effective at improving AMNT performance. Fig. 5.4 shows the performance normalized to the volatile secure memory setting for all the approaches for the three pairs of multiprogram workloads (including prior work and AMNT and AMNT++). AMNT++ counteracts the multiprogram behavior that impacts the efficacy of AMNT. For example, the *bodytrack* and *fluidanimate* workload display an example when applications may impact their mutual spatial locality. In this scenario, AMNT++ can make a difference by reordering the physical pages accordingly and increasing subtree hit rate from 91% to 97%. As a result, AMNT++ performance overhead is reduced from 8% (AMNT) to less than .1% compared to leaf persistence (the best performing approach). Note that the performance overhead of AMNT in the single program experiments for *bodytrack* and *fluidanimate* are less than .1% and 2.1% respectively.

The *swaptions* and *streamcluster* workload and the *x264* and *freqmine* workload are not memory intensive, and as a result the performance overhead is negligible across both approaches. One could imagine a theoretically adversarial case in which the subtree bounces back and forth between subtree regions, especially with distinct address spaces for multiple processes. This evaluation finds that these cases do not occur in practice. In the study of single program workloads, the analysis finds that the subtree root moves 0.3% of

data accesses on average (3 subtree root movements for every 1000 data memory accesses). In multiprogram workloads, this occurs 0.1% of data accesses, which implies no significant difference in memory behavior.

5.7.3 Subtree Sensitivity Analysis

As described in Section 5.5, the AMNT subtree level can be configured in the BIOS so that the hardware can be modified for varying workload characteristics. To build an intuition about these workload characteristics, this evaluation presents a sensitivity study of storing the subtree at varying levels under the multiprogram workloads. As these configurations are intentionally designed to constrain the efficacy of AMNT, this section demonstrates the efficacy of AMNT++ at changing application behavior to improve subtree hit rates.

Fig. 5.6 shows performance impact of varying the AMNT subtree root level with and without the modified operating system (AMNT++ and AMNT), and Fig. 5.5 shows the subtree hit rates for the same configurations. As the subtree root level increases (i.e., moves closer towards the leaves), it protects less data thereby constraining its efficacy. For example, in the *bodytrack and fluidanimate* workload, the subtree hit rate improves by at least 5% when the subtree root is placed between levels 3 and 7 with AMNT++ compared to AMNT. On the other hand, workloads like *swaptions and streamcluster* and *x264 and freqmine*, the runtime performance is not bound by the secure memory hardware and as a result the performance impact is less evident. This result demonstrates that AMNT++ is able to improve the hot region tracking of the underlying hardware without needing to complicate the hardware in a multiprogram environment.

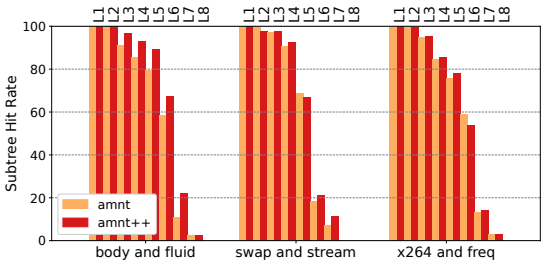


Figure 5.5: Subtree hit rates for multiprogram PARSEC workloads varying AMNT subtree level.

5.7.4 The Cost of AMNT++

In traditional systems evaluation, modifying the operating system can be viewed as an extreme measure as it impacts all applications running on the system, and may incur unavoidable overheads throughout the system. Given that AMNT requires new hardware, the concern over potential adoption is less pertinent, as using a modified operating system on a new system is less burdensome. This evaluation uses the PARSEC multiprogram workloads with and without the operating systems modifications in AMNT++ to evaluate the runtime overhead of the modified operating system and quantify how intrusive it is. To perform this evaluation, this work uses the single program and multiprogram configurations where appropriate.

Table 5.2 describes the impact of the modified OS on application behavior. The normalized performance column reflects the number of cycles to run the multiprogram workload with the modified OS over the number of cycles with the unmodified OS. The impact of the modified OS is negligible on the overall performance. This result is due to the fact that the physical reclamation of pages is an infrequent operation, and the modifications are mostly transparent to the progress of the application.

On the other hand, the number of additional instructions of the OS modification is relatively small. The second column of Table 5.2, the instruction overhead, reflects the additional number of instructions in

the modified OS compared to the unmodified OS. On average across the PARSEC benchmark suite, the instruction overhead of AMNT++ is 1.96%.

Given that the number of additional instructions are small and the impact on performance overhead is small, any speedup of AMNT++ is due to the increased locality of the application, improving the overall efficacy of the on-chip cache hierarchy. The reason for the additional instructions not impacting performance is that the reclamation process is typically off the critical path (i.e., when physical pages are reclaimed) by design.

5.7.5 Multithread Analysis

This evaluation of AMNT is performed with the SPEC CPU 2017 benchmark suite [41] to perform multi-threaded analysis consistent with prior art [317, 301, 8, 88, 24]. The evaluation uses the speed benchmarks with ref inputs, and it fast-forwards to a region of interest as determined by SimPoint [205] in the benchmark before simulating 500 million instructions. As done in prior work, this evaluation uses a four core simulation with a 32kB data L1 cache, 48kB instruction L1 cache, 512kB L2 cache, and 8MB L3 cache.

Fig. 5.7 shows the normalized cycles of the SPEC CPU 2017 benchmarks over a volatile secure memory system which does not account for the persistent state of the metadata. AMNT reduces runtime overhead by as much as 41% and by 13% on average compared to the state-of-the-art, Anubis [317]. Osiris [301] (not shown on the figure) results in about the same runtime overhead as leaf persistence as its design prioritizes runtime over recovery time. However, Osiris' recovery time degrades as memory capacities increase, making it a bad option for cloud service providers who are unable to withstand hours of recovery time for the large capacities of SCM typically used in datacenters. Compared to the leaf and strict persistence baselines, AMNT has a runtime overhead of less than 2% compared to leaf persistence, and up to an 8× reduction in overhead relative to strict persistence (shown in Fig. 5.7).

AMNT has the biggest impact on write-intensive applications. Write-intensive workloads (e.g., *xz*, *lbm*, *deepsjeng*) suffer from the strictest persistent mechanisms, as they place writes on the critical path of application execution. For *xz*, the most write memory intensive benchmark, AMNT results in 32% runtime overhead while Anubis has 41% overhead and BMF has a 7× overhead. AMNT reduces the runtime overhead as it uses leaf persistence semantics on the hot regions of the programs, while keeping the recovery time bounded to a predefined amount. Read-intensive applications are largely optimized by volatile on-chip caches and are unaffected by the metadata persistence model. However, for mechanisms that add complex calculation for memory reads (such as Anubis and BMF), the persistence model still adds to the runtime overhead. For example, AMNT exhibits negligible overhead versus leaf in *cactuBSSN* and *mcf* because they are mostly read memory-intensive benchmarks. Yet, Anubis and BMF both have significant overhead. Anubis suffers from costly metadata cache misses and BMF simply resembles the behavior of the strict performance protocol resulting in high performance overhead. In contrast, AMNT improves performance overhead as it optimizes metadata cache behavior.

Table 5.2: Impact of the modified operating system in multiprogram workloads.

	Normalized Performance	Instruction Overhead
body and fluid	0.992	1.004
swap and stream	0.967	1.021
x264 and freq	1.013	1.01

Table 5.3: Hardware overheads of the state-of-the-art for a 64kB metadata cache. Note that BMF overheads is metadata cache size dependent and it requires an additional 6 bits of volatile capacity per cache line.

	NV On-Chip	Vol. On-Chip	In-Memory
BMF	4 kB	768 B	-
Anubis	64 B	37 kB	37 kB
AMNT	64 B	96 B	-

5.7.6 Hardware Overhead

Spatial overheads for secure memory should be minimized as on-chip area is in high demand for various hardware optimizations across a multitude of workloads. For example, if secure memory hardware occupies on-chip space for the LLC, then the application will incur more LLC misses and be further bound by secure memory. Furthermore, applications are becoming more memory intensive, placing more emphasis on the importance of caching values in the larger components of the chip (*i.e.* LLC). Finally, trends in secure memory have moved towards reducing the in-memory spatial overhead of secure memory, so storing more security metadata in memory is undesirable [298, 224, 256].

This evaluation compares the on-chip volatile and non-volatile overheads separately as these may be composed of different technologies (SRAM vs. Flash). The hardware area overheads for Anubis, BMF and AMNT are listed in Table 5.3. This work assumes that the on-chip root of the BMT must reside in on-chip Flash to conform to the threat model which keeps the root on-chip at all times for all three mechanisms. In BMF, the non-volatile on-chip space consists of an additional non-volatile metadata cache used to store the subtree roots (4kB by default in that work). The volatile on-chip space requires an additional 6 bits per cache line for the frequency counters in addition to the metadata cache. For a 64kB metadata cache, which holds 1024 64B cache lines, the frequency counters amount to 768B.

In Anubis, the non-volatile on-chip space is occupied by the additional root required to track the shadow Merkle Tree (64B). The volatile on-chip space is composed of both the metadata cache and, optionally, the shadow MT cache (37kB).

Area overhead in both of these prior works is a function of memory size. In BMF, a bigger metadata cache will result in more space for frequency counters, or new workloads might demand more subtrees to be tracked for performance. The implementation of Anubis include the shadow table in a distinct on-chip cache for a 64kB metadata cache to remain consistent with [109], and it models the non-volatile metadata cache in BMF as a distinct on-chip cache with a non-volatile metadata cache proportional to the volatile metadata cache size (4kB).

In contrast, one of AMNT's goals is to limit the additional hardware components both on-chip and in-memory. Like Anubis, AMNT has an additional non-volatile register on-chip to track the root of the fast subtree. The volatile on-chip space includes a 768-bit history buffer, and the metadata cache. The 37kB volatile shadow cache in Anubis is much bigger in capacity than the volatile history buffer required for AMNT. The 4kB non-volatile subtree root cache in BMF requires a significantly greater non-volatile on-chip resource compared to the single non-volatile on-chip register required in AMNT to store the subtree root. A direct consequence of having a software and hardware co-design, such as AMNT++, is that the resulting area requirement is low. On-chip area is in high demand for various hardware optimizations across a multitude of workloads, so area overhead of this approach is minimized. As such, AMNT achieves its design goal of limiting additional hardware components on-chip and in memory. In addition, as discussed

Table 5.4: Recovery times (in ms) for the different protocols as a function of memory size.

	2.00TB	16.00TB	128.00TB	BMT stale %
leaf	6,222.21	49,777.78	398,222.21	100%
strict	0	0	0	0%
Anubis	1.30	1.30	1.30	fixed
Osiris	50,666.67	405,333.32	3,242,666.64	100%*
BMF	0	0	0	0%
AMNT L2	777.77	6,222.21	49,777.78	12.5%
AMNT L3	97.22	777.77	6,222.21	1.56%
AMNT L4	12.15	97.22	777.77	0.2%

in the next section, AMNT's performance is agnostic to other features, such as metadata cache size and memory size.

5.7.7 Recovery

The recovery process requires both the fetch of counter values from memory and the computation of the hashes of data-independent regions. For example, nodes within a level (i.e., siblings or cousins in a BMT) are data independent, however since a parent node cannot be computed without knowing the value of its children, parents and children have a data dependent relationship in hash recomputation. To relieve this data dependency, the re-computed hash values for a level are written back to memory before the next level can start the hash computation. Seeing as the hash computation is both fast and pipelined, this work assumes that the recovery time is bound by the memory bandwidth. Note that the split of reads and writes in the recovery workload is a ratio of 8:1 (reads:writes) as eight children are to be fetched in order to compute a parent hash (which will be written back to memory). A single Optane DIMM supports around 4 GB/s of total bandwidth when subjected to this mixed read/write sequential workload [123], of which around half of this bandwidth (2 GB/s) is dedicated to reads. Assuming a six-channel machine [127, 113], this provides a total read bandwidth, at recovery, of 12 GB/s to memory, which is the essential performance bottleneck for recovery. This work uses this bandwidth to generate the data in Table 5.4, which shows the time it takes to recover each of the baseline and state-of-the-art configurations after a system failure.

Unlike prior approaches, recovery time in AMNT scales with the level in which the subtree root is placed and is reconfigurable. For example, with the AMNT subtree root configured at level 3 of the BMT, it has a slower recovery time than Anubis [317] (as shown in Fig. 5.4). Although slower than Anubis [317], recovery times are still reasonable and much faster than other state-of-the-art alternatives. In the event that a service provider cannot tolerate long periods of downtime, AMNT can be re-configured with a subtree root closer to the leaves. For instance, with the subtree root configured at level 4 for a 2TB memory the recovery time is 0.01 seconds (see Table 5.4).

5.8 Related Work

Other works that have proposed using a fast subtree can be classified into two works that leverage indirection [9, 284] and those that use data addresses to determine its membership or non-membership in the fast subtree [214]. AMNT uses data addresses to determine membership in the fast subtree protocol. This design choice is an advantage over approaches that use indirection for two reasons: (1) approaches that leverage indirection cannot begin until some information is fetched that determines which authentication protocol to

use, and (2) the logic controlling this indirection leads to significant on-chip area and in-memory storage overheads (additional caches, in-memory queues, etc. [113, 215]). Assure describes a protocol where a single subtree is tracked to reduce the authentication and update path length for frequently accessed data [214]. However, this work does not target multiple persistence strategies. AMNT intentionally addresses this non-trivial question, and results in performance benefits.

5.9 Conclusion

Storage class memory (SCM) offers high density, non-volatile storage with dramatically faster speeds than traditional storage systems. However, this non-volatility creates new security challenges. This chapter presents *A Midsummer Night's Tree* (AMNT), a novel hybrid persistent Bonsai Merkle Tree (BMT) protocol for integrity-protected non-volatile SCM. AMNT improves performance overhead by up to 41% compared to the state-of-the-art approach while providing fast and configurable recovery times that are a function of the level of the subtree root rather than the memory size.

AMNT provides a strong basis for considering how, by writing software directly for some architecture, performance may be globally benefited. However, AMNT faces certain limitations predicated on this feature. ① AMNT++ will struggle to handle cases of fragmentation. If there are few physical addresses available to allocate, then it is unlikely that a biased allocation procedure will benefit performance. ② Though out of scope of the threat model in this work, the modified operating system introduces slow paths through normal execution that an adversary can hijack to perform denial-of-service attacks. This work argues that an adversary within this threat model would be able to perform such an attack anyways, but the new operating system provides an adversary an additional surface to do so. ③ It is worth noting that the issue of long recovery is only an issue at extraordinarily large capacities. As a result, it may be reasonable in many commodity cases to use a leaf persistence scheme to maintain metadata or even a further relaxed scheme like Osiris [301]. Prior work indicates that the issue of recovery may be a bigger problem than it is given unrealistic assumptions about the recovery workload in practice (i.e., Anubis [317] makes an unsupported claim that it takes 100ns to recover a node without considering parallel work). The analysis in Table 5.4 better characterizes this problem.

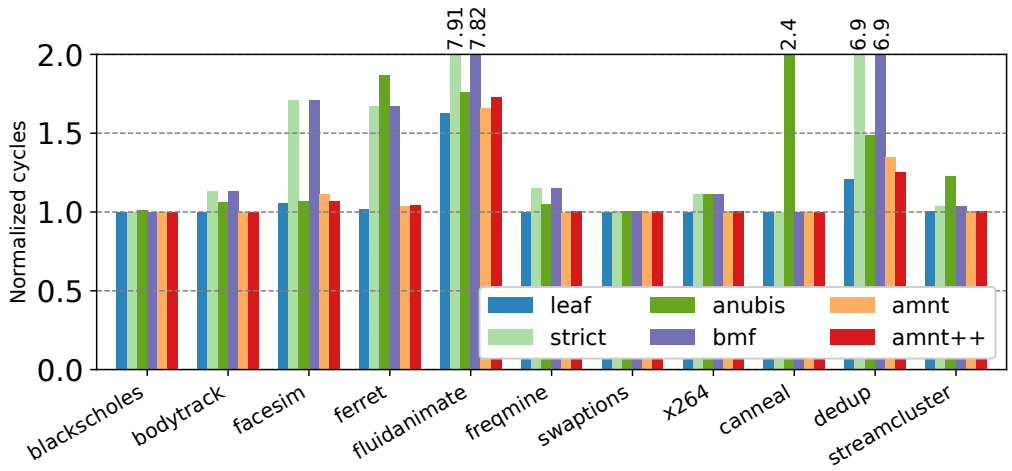


Figure 5.3: Normalized cycles in single program PARSEC workloads.

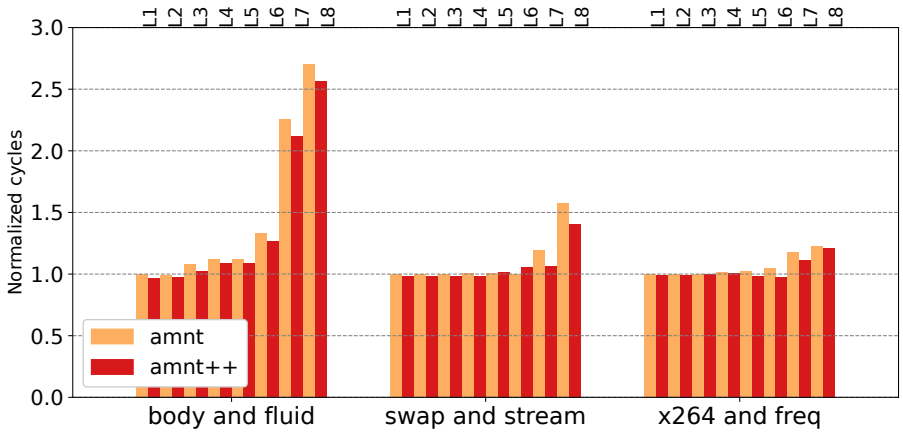


Figure 5.6: Normalized cycles in multiprogram PARSEC workloads varying AMNT subtree level.

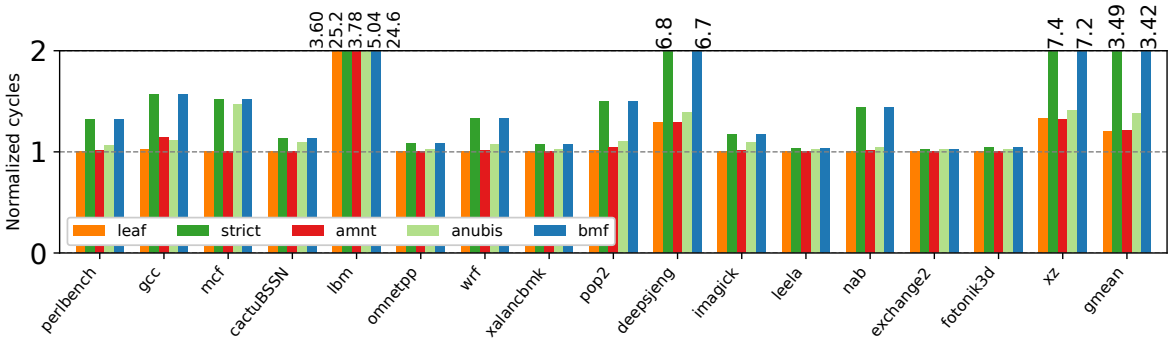


Figure 5.7: Runtime comparison of AMNT, Anubis, and BMF protocols for the SPEC 2017 CPU benchmarks normalized to writeback secure memory protocol. Lower is better.

Chapter 6

CAPULET: Cache Pooling Metadata Caches in Secure Disaggregated Memory Systems

6.1 Problem Statement

As described in Chapter 5, an emerging problem in modern workloads is the inability of local memory to meet the capacity and performance demands of modern workloads. This chapter is predicated on disaggregated memory, an alternative emerging technology to address this problem. Disaggregated memory architectures architectures (e.g., CXL [67], NVIDIA's NVLink [200], and UALink [268]) describe a system through which memory devices are connected to one or more processors via I/O. This configuration enables a processing element to distribute its memory accesses across memory devices to avoid saturating the available bandwidth of an individual device. As such, these memory systems are attractive for cloud deployment [155, 31, 101, 158] given their ability to implement local memory expansion [5, 132, 250], realize cross-host memory pooling [155, 98, 282, 11], and distributed shared memory [311, 259, 296, 162]. Unlike NVMs, which promise a denser memory to better utilize the local capacity of a motherboard, a disaggregated memory system promises scalability via remote, I/O attached memory devices. As a result, disaggregated memory systems are not bound to the limitations imposed by a motherboard, such as capacity restrictions due to a limited number of DIMM slots. These memory systems are attractive for cloud deployment [155, 31, 101, 158] given their ability to implement local memory expansion [5, 132, 250], realize cross-host memory pooling [155, 98, 282, 11], and distributed shared memory [311, 259, 296, 162].

A disaggregated memory system also allows an application to exploit higher bandwidth capacity than what can be afforded by local memory. At a high level, the remote address space is exposed to the operating system, but the management of these addresses is handled by the architecture. The process of utilizing and managing a disaggregated memory system is detailed in Sec. 6.3. Therefore, the architecture can partition an address space across memory devices to avoid saturating the available bandwidth of an individual device. Instead, the ideal peak memory bandwidth of a disaggregated memory system is the minimum of the interconnect bandwidth and the sum of the device bandwidths.

Unfortunately, the memory vulnerabilities described in Chapter 2 are a function of the device technology regardless of whether they are deployed locally or remotely. What's worse, many proposed disaggregated memory system deployments consider schemes in which multiple processing hosts will share memory devices. This increases the exploitable surface for an adversary. Therefore, protecting the state of data in a remote memory device remains a pertinent problem. Unlike in a local memory, a disaggregated memory system does not give a host processor a complete view of the memory system. However, the processor must still authenticate untrusted data prior to loading it on-chip. Otherwise, the host processor must expand its trusted computing base to include some other authenticating entity. The implications of this are described in Sec. 6.4. As a result, key to the issue of securing a disaggregated memory system is the issue of managing the metadata associated with a processor's remote memory.

6.2 Introduction

Given the presence of these attacks, cloud providers must deploy secure hardware devices so that developers can reason about the safety of their deployed applications. In particular, the presence of the aforementioned memory vulnerabilities provides strong motivation for the development and deployment of *secure memory* [248, 298, 99]. As described in Chapters 2 and 3, performance is a critical problem for secure memory devices [80, 112, 169]. Chapter 3, explores the scalability of the *metadata cache* to optimize the secure memory protocol. The benefits of the cache are twofold: ① security metadata stored in the cache can be accessed at lower latency than those in memory, and ② on-chip caches are not susceptible to the same physical attacks as off-chip memory devices, so accesses that hit in the cache can serve as roots of trust for authentication without needing to fetch metadata all the way up to the Merkle tree root, significantly reducing bandwidth pressure on the underlying memory device. These two benefits are substantial, and using a metadata cache has been demonstrated as offering orders of magnitude improvements in runtime [262]. Unfortunately, even with the metadata cache, secure memory has been demonstrated to be too slow in practice and its optimization remains an ongoing, high-priority issue in academia and industry [122, 150, 7, 310, 23].

Note that simply increasing the size of the metadata cache to improve utilization is not scalable. Instead, this chapter considers new ways of distributing the metadata across the memory system to leverage the memory system fabric to achieve a similar effect. Note, Chapter 3 explores a means of modifying the structure of the integrity tree to promote the benefits of the cache without being subject to the cache itself. However, this approach is complex, requires additional metadata, and incurs periods of blocking pending authentications to restructure the tree shape. Furthermore, research concerning memory security for disaggregated memory systems is in its infancy. While early proposals have envisioned storing security metadata in remote memory, these designs do not consider secure memory *placement* and *management*. This is a non-trivial detail that requires more detailed investigation. This chapter contributes a detailed design of secure memory with disaggregated memory.

This chapter sets out to accomplish two goals: 1) describe a secure memory model suitable for disaggregated memory systems, and 2) design a new protocol that leverages the interconnect fabric of disaggregated memories to optimize secure memory. Note that secure memory is largely optimized by the security metadata cache. Improvements in metadata cache utilization equate to performance improvements.

Achieving the first goal requires defining how a processor and a remote memory may coordinate the

maintenance of secure memory metadata. Doing so is non-trivial: candidate schemes face limitations in performance and correctness. This chapter proposes an efficient scheme to store secure memory metadata, where a memory device in a disaggregated memory system maintains the security metadata associated with its addresses that a host can later use for authentication. In addition, a host utilizing the memory system may cache authenticated metadata associated with its remote data in its metadata cache. This scheme reduces the work required for future authentications of remote data.

To achieve the second goal, this chapter proposes *CAPULET*, a protocol that implements cache pooling in underlying external disaggregated memory devices. CAPULET improves the metadata cache utilization of a host in a disaggregated memory system by storing evicted values in metadata caches elsewhere in the memory. To do so, this chapter introduces *cache pooling*, a novel abstraction for caches to coordinate with each other in a disaggregated memory system. Much like a memory pool, a cache pool allows a local cache to expand its capacity by viewing other devices in the system as members of an abstract pool. From this view, the cache can optimistically offload evicted values to the pool and later query the pool for missed values. This expands the "effective" capacity of the individual device, which is of particular importance when the utilization of the security metadata cache is challenged by the strain of the workload or disaggregated memory protocol. By taking this approach, CAPULET improves the efficiency of the secure memory protocol while upholding the design goals of not relying on large local security metadata caches or complicating the logic to cache it. CAPULET can improve secure memory performance by 15% on average and by as much as 4× for high performance computing (HPC) workloads.

This chapter makes the following contributions:

1. It provides an overview of various protocols to implement secure memory in a disaggregated memory system.

2. It introduces *cache pooling*, a novel abstraction model to reason about memory device caches in disaggregated memory systems.

3. It demonstrates the advantages of cache pooling by evaluating the design on two disaggregated memory system deployments: local and in the cloud.

6.3 Background

6.3.1 Disaggregated Memory

Application demands of memory are ever increasing. To account for the increased demands in capacity and bandwidth, disaggregated memory technologies (i.e., CXL [67], NVLink [200], UALink [268], etc.) have emerged. At a high level, these technologies describe an I/O protocol that allow memory devices to be attached to one or more hosts. For simplicity, this discussion focuses on the Compute Express Link (CXL) interconnect as its standard is open source by design[1]. The communication system enables high global memory bandwidth with relatively low latency accesses due to the underlying interface (e.g., CXL uses PCIe [132], whose sixth generation supports 64GT/s per lane [234]). By taking this approach, disaggregated memories are highly flexible in their deployment and can be used to implement a variety of memory systems.

[1]This chapter uses the terms disaggregated memory, remote memory system, and CXL interchangeably.

These memory systems can be used to expand the capacity of a single computing host [5], dynamically allocate remote memory resources across several hosts [143, 98, 11, 135, 31, 155], and/or enable remote shared memory among several hosts [311, 259, 296].

A sample configuration of a CXL architecture is illustrated in Fig. 6.1. Seeing as CXL is a PCIe-based protocol, an application interacts with a CXL device through one of several proposed software interfaces. When a root port on the system bus is established for a CXL-capable device, a kernel driver queries the remote device for its base address and internal memory size. Note that in the provided example, the root port (i.e., a CXL card with a CXL controller [166]) may be attached directly to the end point of a CXL memory device or to a CXL switch. In either case, the querying procedure on connection establishment is the same. This allows the operating system to maintain a coherent view of the global physical address space. The operating system may maintain remote memory mappings in an `mmap`-able file associated with a device file [98] or remote memory may be accessible to an application through the zNUMA interface [155]. In both examples, the application is able to interact with memory in a remote CXL device through its virtual address space while the discovery and maintenance of remote memory is maintained by the operating system.

This abstraction allows the host to simplify its view of remote memory. Suppose Host A implements remote CXL memory through an `mmap` interface to device files. In this case, there would be a virtual memory space for each of the root ports: one interfaces directly with the local memory expansion end point and one with the CXL switch coordinating the memory pool. This approach allows the application to explicitly allocate objects and data to a particular CXL device or to interact with "remote" memory in a device-agnostic manner.

To request a virtual address that is mapped to a remote memory, the application contacts the operating system to translate the address. Upon completing the translation, the operating system makes a request of the associated device. To the host processor, this address will appear as a physical address outside of its memory address range. The system bus directs the request to the appropriate root port for the address range containing the request. Then, the appropriate CXL request will be sent to the connected peer to be managed by the remote memory system.

A natural application of this protocol and technology is towards *memory expansion*. That is, if a host's local memory capacity is restricted (i.e., due to limited DIMM capacity on the motherboard), the host may choose to leverage external memory devices via CXL to expand its capacity. These remote memories may be attached to a CXL card directly or via a CXL switch if the number of physical ports is limited. Direct connection between root port and end point requires lower latency than CXL switch (55ns point-to-point latency as compared to 270ns [155]). Given that the remote memory is I/O attached, the maximum capacity of remote CXL memory is theoretically unbounded.

Beyond local memory expansion, *memory pooling* has been proposed as an application of disaggregated memory systems. A memory pool [64, 199, 74, 155, 135, 143] describes an arrangement of memory devices that are accessible to a variety of hosts without their explicit knowledge of which device they are accessing. That is, if a host wants to access some "physical address" that reside in a memory pool, the hardware in the disaggregated system (i.e., a CXL switch) manages any additional mapping of that address to the appropriate device [119]. Memory pooling enables the hardware to implement properties like replication and data partitioning so as to minimize application complexity. Cloud providers can leverage memory pooling architectures to elastically improve utilization of memory resources, such that stranded memory can be reallocated to other processes (i.e., serverless workloads) [155, 153].

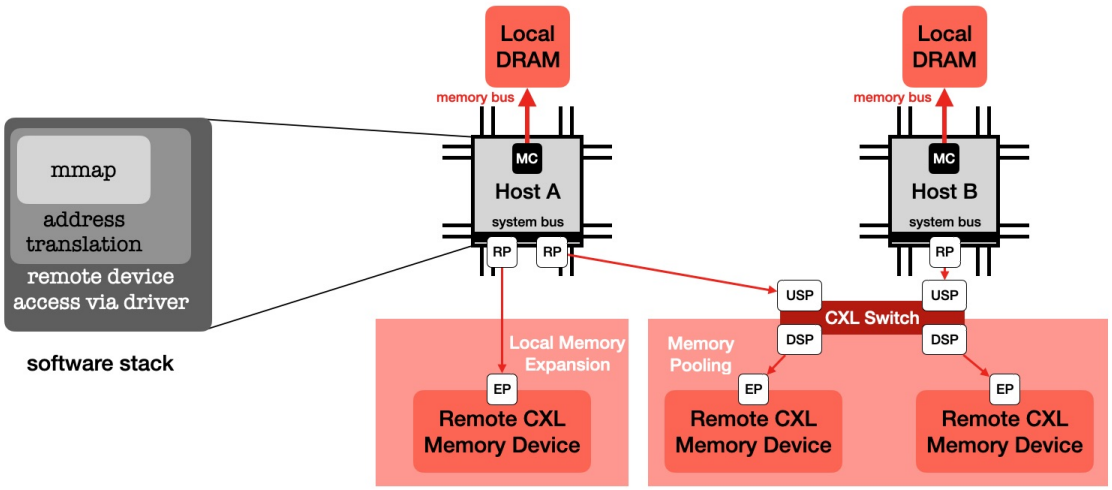


Figure 6.1: Sample CXL architecture proposed in [98]. Hosts maintain root ports (RP) on their system bus that attach to CXL end points (EP) or user-side ports (USP) in a CXL switch. A CXL switch may attach to a CXL device EP via a data-side port (DSP). Memory expansion can be implemented via direct connection between host RP and a CXL EP. Memory pooling may be implemented with a routing table in the CXL Switch.

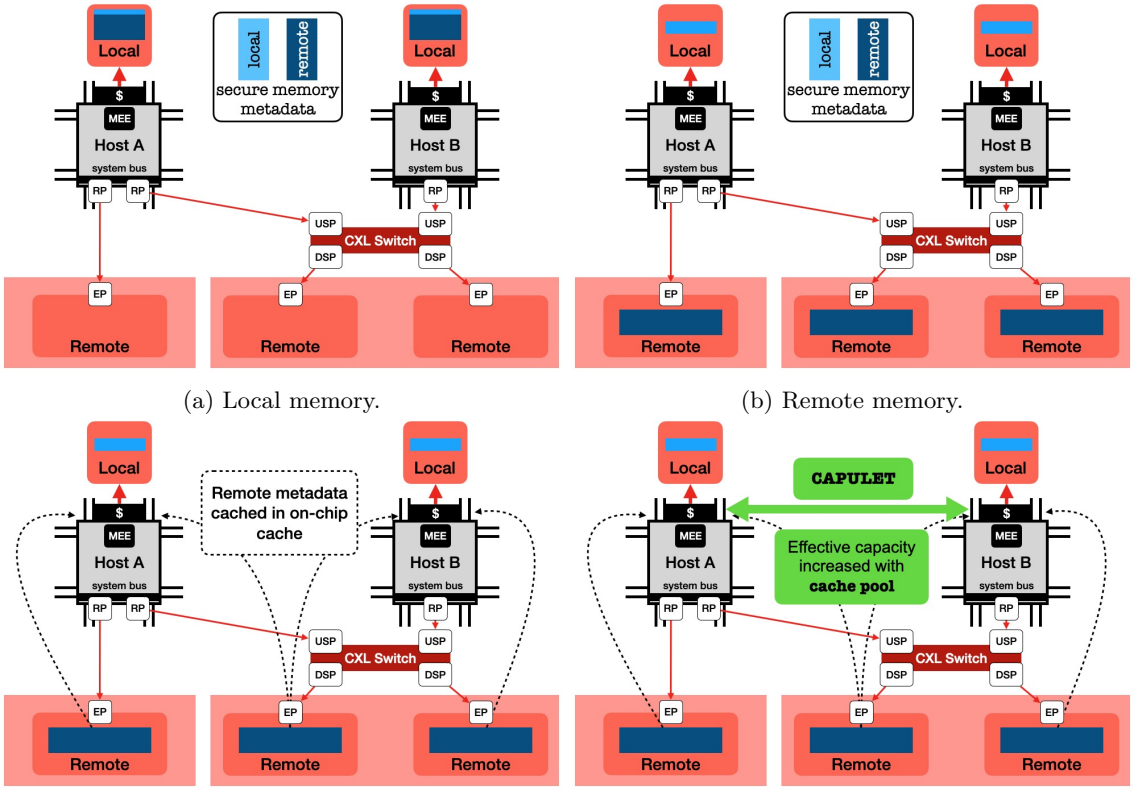
CXL hardware can manage the mapping from a host's view of a physical address to an address in a device. CXL has been proposed as a means for multiple hosts to use remote memory as a *distributed shared memory* [259, 311, 296]. The CXL switch allows multiple hosts to access the same location in a remote memory device. Doing so allows an application running on multiple hosts (i.e., a distributed system) to use memory as a point of coordination as opposed to coordinating via the network.

6.3.2 Related Work

There have been several proposals for how to secure memory in a disaggregated environment in prior literature. In particular, prior art has considered how to build a single root of trust over all addresses on remote devices [10, 85, 238]. All remote values are considered untrusted, and any memory controller is responsible for protecting the state of all other data. In this scheme, the key challenge is ensuring that the root stored on any trusted processing element can maintain a coherent view of the true root of the entire state of remote data. Processing elements must coordinate in order to verify that legitimate memory operations from other hosts are reflected in the roots of trust in all other processing elements.

6.4 Metadata Placement

Local and remote memory devices alike are subject to physical corruption. Protecting data stored in these devices with secure memory is critical. The host processor and memory system must maintain the metadata to implement encryption and integrity, and the host processor must decrypt and authenticate the data after it is fetched. However, there are non-trivial implications of the secure memory metadata placement in a disaggregated memory system.



(a) Local memory.

(b) Remote memory.

(c) Cached on-chip.

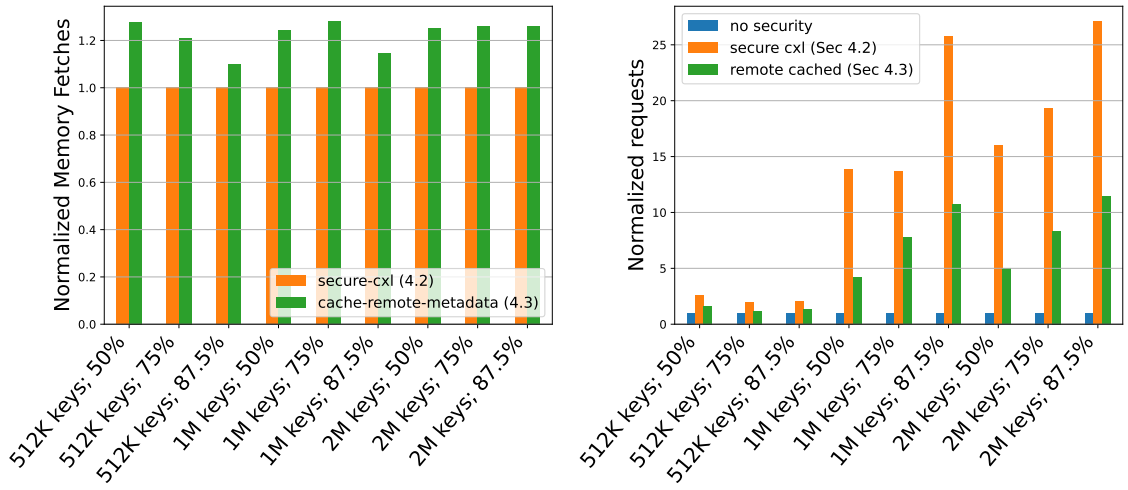
(d) Increased cache capacity with CAPULET.

Figure 6.2: Schemes to maintain metadata for remote address space in disaggregated memory system.

6.4.1 Maintaining Remote Metadata Locally

One scheme to implement a secure disaggregated memory would be to consider local and remote memory as a single address space for which all of the associated metadata is maintained in the host's local memory. In this scheme, depicted in Fig. 6.2a, the host will reserve and initialize metadata for the remote address space in its local memory upon initializing the connection with the remote memory system. The encryption counters for local and remote data can be protected by the same integrity tree which allows the host to protect local and remote memory with a single root. When the host requests local or remote data from memory, it sends requests to local memory for the metadata at the associated address. This allocation allows data fetched from the disaggregated memory system to be authenticated by the same root of trust (i.e., BMT root or metadata cache hit) as the host's local memory.

Unfortunately, this organization of security metadata faces the issue that the capacity of remote addresses is theoretically *unbounded*. Given that each piece of data in memory has its own associated metadata, the size of the metadata (typically 12.5% of protected memory [222, 261]) is similarly potentially unbounded. Storing all metadata addresses in local memory is impossible if the global metadata size (i.e., metadata for local and remote memory) exceeds the capacity of local memory. Even without exceeding the capacity, storing all metadata in local memory leads to suboptimal performance. Every byte reserved for the storage of metadata comes at the cost of space for application data. When writing an application for a tiered memory system (i.e., disaggregated memory), accessing local data should be prioritized as much as possible [37].



(a) Authentication overhead shown as normalized overall memory fetches (i.e., metadata cache misses) for secure memory metadata.

(b) Interconnect communication overhead of secure remote memory normalized to a non-secure remote memory with a partitioned skip list.

Figure 6.3: Analysis of secure memory in disaggregated memory systems with various metadata placement schemes.

Observation: if the remote address space is unbounded, then the remote metadata size is unbounded.

6.4.2 Maintaining Remote Metadata Remotely

Storing all metadata locally leads to the takeaway that an unreasonable percentage of the local memory may be reserved for security metadata. To account for this limitation, an alternative scheme can store security metadata for local addresses in local memory, and security metadata for remote addresses in remote memory. Such a scheme is depicted in Fig. 6.2b. When the host establishes the connection to a remote memory device, the host initializes and requests the storage that the associated security metadata be stored in those devices. While a host may view remote memory as a single address space, it may be composed of multiple devices in practice. As the host initializes the secure memory metadata to be stored at a remote address, the disaggregated memory system interconnect fabric (i.e., a CXL controller or switch) manages the mapping of these addresses. That is, the remote memory metadata may be stored across the remote devices that appear as a single address space to the host, so the host only needs to maintain a single integrity tree for these addresses.

The host processor is still responsible for *authenticating* the remote data with the remotely stored metadata. To do so, it should maintain an additional root for the remote integrity tree in its root of trust. When the host processor makes a request for data stored in a remote memory device, the host processor will issue the data request and the corresponding requests for metadata to the fabric of the memory system. The remote memory system will issue the requests according to how the addresses are mapped in the system. Once the host processor starts receiving the corresponding metadata, it can begin the data authentication process against the root of trust for remote memory just as if the data were stored in local memory. With this approach, the amount of security metadata is a function of the capacity of the protected memory, which resolves the limitation from Sec. 6.4.1.

Observation: maintaining metadata remotely requires significant communication bandwidth.

To demonstrate this effect, this work implements the CXL partitioned skip list depicted in Fig. 6.4 in CXL-DMSim [289] (full evaluation details in Sec. 6.6) to measure the number of requests communicated from remote memory to the host. A skip list describes a widely deployed non-blocking, concurrent key-value store with $\log n$ `insert`, `remove`, `lookup`, and `update` operations [66, 65, 22]. Nodes have references that "skip" ahead to later parts of the data structure, and the probability that a node has n references is $\frac{1}{p^{n-1}}$, where p is typically 0.5. As a result, the proportion of data in the lowest n levels of the structure is $1 - \frac{1}{p^{n-1}}$ (i.e., 50% of the data in the lowest level when $p = 0.5$). The data structure lends itself to partitioning across memory regions and has been well studied in the context of NUMA [68, 263, 42].

The evaluation of the skip list is clustered by the number of keys in the keyspace and the percentage of the data structure in remote memory. Fig. 6.3b shows the communication overhead. This experiment shows that implementing secure memory over remote addresses can increase the number of CXL requests (for data and metadata) per instruction by up to 27× compared to a non-secure memory. This overhead is a function of the additional work required to maintain the metadata associated with the remote address space. This trend is directly related to application size, as communication overhead gets worse as the size of the skip list increases. Additionally, when a greater percentage of the data structure is in remote memory, there are more requests that need to be authenticated, and thus communication overhead increases dramatically.

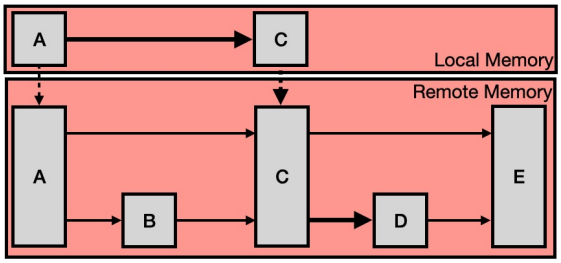


Figure 6.4: A skip list partitioned for disaggregated memory. Searches start in the upper levels (local memory) and may continue towards lower levels (remote memory) until value is found. Emphasized references demonstrate a search for `D`.

These findings are consistent with prior work, which finds that communicating all metadata relevant to some remote data can come at steep performance costs. This phenomenon is well studied in the context of securing CPU-to-GPU memory [3, 309, 308, 310, 2]. Each transmission consumes bandwidth along the interconnect for the disaggregated memory system. This phenomenon limits the practicality of such an approach.

6.4.3 Caching Remote Metadata Locally

To account for the communication overhead of the prior approach, this analysis considers caching remote metadata locally in the on-chip metadata cache, shown in Fig. 6.2c. Recall that the metadata cache serves as an important optimization to alleviate the metadata bandwidth by reducing the number of integrity tree fetches. By caching remote metadata, a similar benefit can be realized towards alleviating the pressure on interconnect traffic. Like in the traditional system, when the host has a metadata cache hit, it stops traversing the integrity tree, reducing remote accesses.

In this metadata placement scheme, the host checks its metadata cache state for a node in the integrity tree path closest to the data (i.e., towards the leaves) when making a request for some remote data. Upon finding such a node (which acts as a root of trust), the host can request only the data and integrity tree

path up to this node. Cache hits result in significantly fewer requests than the full leaf-to-root path. After the metadata block is fetched from the remote memory and authenticated against the root of trust in the host, it can also be placed in the host's metadata cache to serve as root of trust for future authentications.

Observation: caching remote metadata in the on-chip metadata cache alleviates interconnect bandwidth pressure.

Caching remote memory security metadata reduces traffic by up to 3.5× due to the reduced metadata fetches required to authenticate against a trusted value. Fig. 6.3b shows the benefits of caching remote metadata on CXL communication traffic. This approach is similar to using a metadata cache for local memory. However, secure memory still incurs up to 11× interconnect communication overhead relative to non-secure memory due to maintaining remote metadata.

Observation: caching remote metadata degrades the efficacy of the metadata cache.

Caching remote metadata in the local metadata cache is not without consequence. Fig. 6.3a shows the number of memory fetches per authentication (i.e., metadata cache misses). That is, hitting at the lowest level of the integrity tree would imply no memory fetches. The analysis shows that caching remote metadata leads to 27% more memory fetches to authenticate *local* data despite the fact that less than 1% of memory fetches in the skip list workload go to remote memory. This is because the metadata cache capacity is constrained, and local accesses in this workload (and across workloads [150, 317, 310]) exhibit metadata locality. Therefore, caching remote metadata introduces tension between interconnect communication overhead and metadata cache utilization.

6.5 Design

This section describes CAPULET, a protocol to manage secure memory metadata caches in disaggregated memory architectures. CAPULET benefits metadata caches under strain by storing security metadata coherently in remote metadata caches in external hosts that share a disaggregated memory system. In doing so, CAPULET addresses the restrictions imposed by the limited metadata cache capacity in a single host. The ideas proposed in CAPULET leverage the fabric of the disaggregated memory system to facilitate the expanded "effective" capacity of metadata caches.

6.5.1 Architecture Overview

CAPULET is built on top of a disaggregated memory system, such as CXL [67], NVLink [200], or UALink [268] in which multiple hosts utilize a disaggregated memory system to implement remote memory. This may be for capacity expansion, as a memory pool, or to implement a distributed shared memory. At least two of the hosts in the system must implement secure memory with a MEE over its local memory (and by proxy utilize a metadata cache). Hosts send memory requests to the memory system via PCIe (consistent with the CXL standard), and the data stored on these devices is untrusted as data in a memory device is subject to corruption. For simplicity, this work assumes that remote address ranges are static. Dynamically sized remote memories, such as serverless memory pools, are out of scope. Upon initialization of the memory

system (i.e., during boot), the secure memory metadata is established in all devices with memory (both hosts and remote memory devices). The hosts participating in a memory pool using remote devices perform an initial attestation to establish a channel for secure communication during the session of interactions [176, 56]. This allows the hosts to securely communicate security metadata to remote memory devices.

This work assumes an architecture in which memory devices are only responsible for maintaining the metadata associated with their own address space. Each underlying remote memory "owns" its address space and bears the responsibility for responding when a request across the communication channel comes in for an address it owns. Note, a host's view of "remote memory" may comprise multiple physical memory devices, so ownership of an address space is enforced by the fabric of the memory system (i.e., a CXL controller or switch). The hosts utilizing this remote memory will not trust remote memory devices to perform any authentication, so hosts making a request for some remote data also request the associated metadata to authenticate its state in its own secure memory hardware. This work assumes that hosts can cache metadata for remote data in their local cache state (the metadata placement scheme from Section 6.4.3). The host only needs to request metadata from the remote memory up to the first metadata cache hit. To update some remote data, the host communicates the new data state to the disaggregated memory system, which updates the associated metadata. Seeing as CAPULET implements a tree of counters, the host updates its trusted counter state (either the integrity tree root or an intermediate cached node) and its associated hash locally[2]. When remote metadata is evicted from the host's metadata cache, it writes it back to its remote location (as in traditional systems).

This work makes two assumptions about the fabric of the disaggregated memory system to implement CAPULET. First, it assumes that there are secure, private channels, through which two hosts can query each other's secure memory metadata caches. This secure communication between hosts can be established using an initial attestation on system configuration. This process is thoroughly described in the CXL standard [67]. The second assumption is that the fabric of the distributed memory system supports broadcasts across the interconnect. This assumption is important to implement the coherence of metadata between hosts. Such broadcasts can be implemented as several private communications from one host to all other hosts or as a request to a CXL switch and the switch sending some communication to other hosts in the system. This work makes no assumptions about the security of these broadcasts, and implementing secure communication across a shared disaggregated memory channel is an orthogonal problem to this work.

6.5.2 Cache Pooling

The next role of this section is to introduce the concept of a *cache pool*. A cache pool, depicted in Fig. 6.5, describes an abstract view of host caches in disaggregated memory. Much like in a memory pool, the cache pool abstraction allows caches associated with other devices in a disaggregated memory system to view the rest of the architecture as an abstract object that can be queried. The cache pool allows host caches to expand their effective capacity. Cache pools allow *all* host caches in the system to access the cache pool as both the querying cache and a member of the expanded capacity. This approach, much like work stealing, allows for better utilization of capacity elsewhere in the system when one device is under more strain.

To implement cache pooling, the cache controller of a pooled cache needs to update its eviction and

[2]If the host implements a BMT-based secure memory, the remote memory devices must respond to the write with the up-to-date metadata in the path so that the host can verify that the root was updated appropriately.

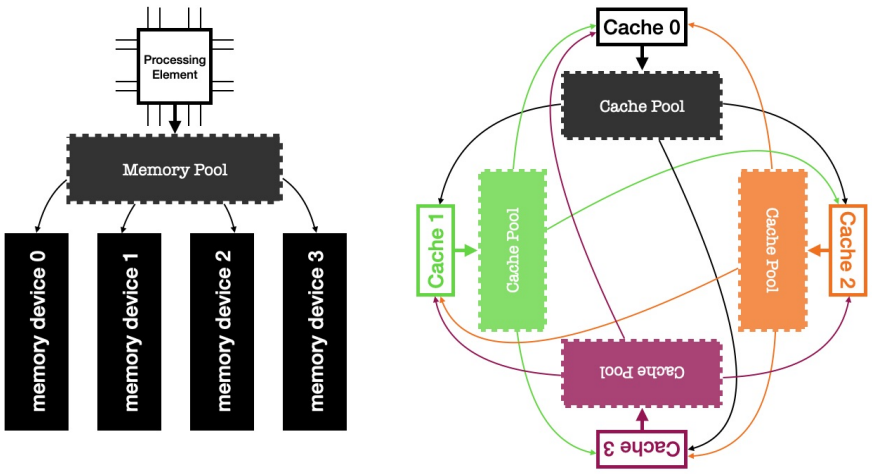


Figure 6.5: A memory pool as compared to a cache pool. In a cache pool, a cache can access the pool of other caches while also being a member of all other pool views.

miss procedures. These updates are depicted in Fig. 6.6. Let's consider the eviction procedure. In step ①, three things must occur in parallel: the evicted data is written back to the lower level of the local hierarchy, the data is written to a temporary buffer on-chip, and a broadcast is communicated across the interconnect to *offer* the value to the pool. The value is not necessarily communicated unless the broadcast is secure, so the temporary buffer maintains the value in the event it should later be communicated. In the context of secure memory, the writeback updates the metadata value in its local memory as before. Upon receiving notification of an eviction, ② an "available" remote cache may choose to accept the offer by privately notifying the offering host that it can accept the value with an acknowledgement. We define "availability" in Sec. 6.5.3. ③ This handshake allows the host to communicate the value from the temporary buffer on-chip to the receiving cache.

In the event that a request misses in a cache belonging to the cache pool, the cache executes its modified miss procedure. In particular, the cache queries both the external caches in the pool and the next level of the local cache hierarchy in parallel (step ④ in Fig. 6.6). In the context of secure memory, the local access will be to fetch the data from memory for untrusted metadata that will require authentication. The miss will also be broadcast across the interconnect fabric to the other members of the cache pool. If the data is found in a remote cache, ⑤ the remote cache will privately communicate that metadata back to the originally requesting cache. After it does this, it must invalidate its cached copy of the data to ensure coherence as it may be updated in the original host.

The host may also have fetched the value from its hierarchy and updated its copy locally prior to receiving a response from the pool. As a result, ⑥ the host must broadcast an invalidation across the interconnect when it dirties a potentially shared value in the pool. More formally, there are two possible schemes to implement coherence if the data cached in the cache pool is shared among multiple hosts. One option is to use the disaggregated memory fabric as a channel to implement the coherence mechanism. Several pieces of work have explored using CXL as a means to implement cross-host coherence [36, 12, 258, 296, 245], but such a scheme comes at the cost of significant traffic on the memory system fabric. Alternatively, the cache pool can mandate that shared data may only be loaded into a host's cache in a read-only state. Shared data in the cache pool is always assumed to belong to some other host. In the context of metadata caches,

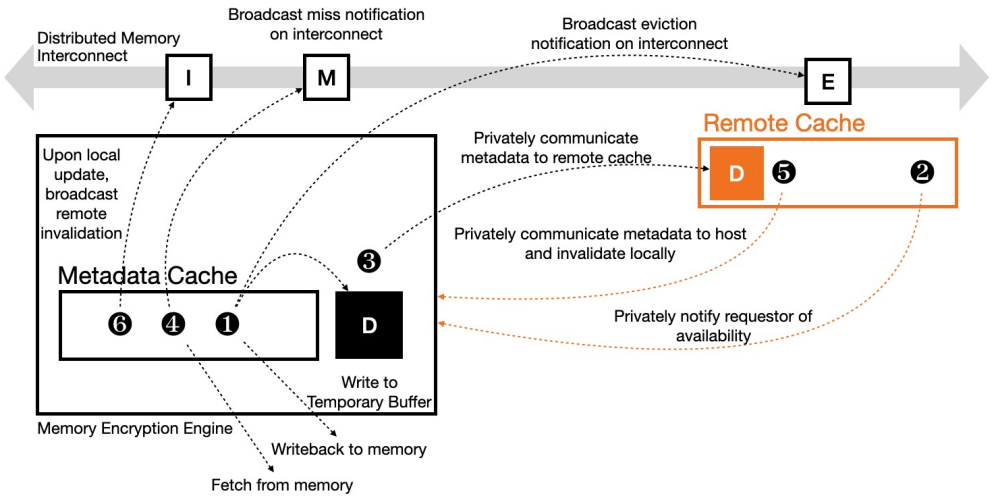


Figure 6.6: Overview of CAPULET with secure memory metadata and two hosts (orange implies cache on a remote host).

the read-only state of metadata can still vastly benefit secure memory performance as it reduces future authentication paths.

Much of the cache pool traffic can be dropped without consequence after some timeout. All requests to the cache pool fall back on an equivalent request to the corresponding memory. When broadcasting an eviction to the cache pool, the state can be dropped if no acknowledgment is received after a timeout. Doing so merely means that the next request for the data needs to be from memory. Furthermore, space reserved to receive the data following an acknowledgment can be freed after a timeout without impacting the protocol.

The cache pool benefits the performance of an individual device at the expense of traffic across the interconnect. Note that this is an important issue, as cross-device coherence of data in disaggregated memory protocols has been limited by the traffic across the interconnect and by the state maintained on the local device. The cache pool similarly increases traffic across the interconnect, and Sec. 6.5.4 describes mechanisms through which CAPULET explicitly addresses this trade-off.

6.5.3 CAPULET

CAPULET takes advantage of cache pooling for secure memory metadata caching, as depicted in Fig. 6.6. Hits in remote caches in the cache pool can be trusted. This trust is established at startup, leveraging attestation features of the devices involved and the Diffie-Hellman exchange protocol for establishing a secure channel [176]. The attestation process can guarantee that the remote host is running trusted logic as it was intended, so it can be assumed that the secure memory protocol is working correctly. From this assumption it can also be derived that the metadata, while residing in the remote pooled cache, is not subject to corruption, cannot be modified by the cache logic, and is only ever communicated securely across the interconnect.

Even though remote cache hits have longer latency than the local hits, the performance benefits are still notable. Hits in remote caches require longer latency, in some cases even comparable latency to main memory accesses, since they need to traverse the PCIe fabric. Prior work notes that access latency to remote devices may be over 100ns in some configurations [155]. However, remote cache hits in CAPULET still

benefit as these values are trusted and reduce the additional memory requests. Further authentication of secure memory metadata against parent nodes is not required after performing an authentication against a trusted value. Thus, remote cache hits provide a performance benefit in terms of reducing the amount of metadata to authenticate as the bandwidth requirement on the device to access metadata is reduced.

CAPULET also allows for remote caches in a cache pool to define when they are "available" to accept remote requests. Given that different hosts may be running different workloads, it is unlikely that both will perform the same memory access pattern in parallel, and one may be more memory bound at some point in time than the others. If some remote cache is under heavy strain, it is better to handle only local requests from the active application primarily running on its address space with the metadata already in its cache. This work defines a node as "available" if it has free space to store the offered metadata or if the metadata from the evicted cache is used more recently by that cache than the eviction target in its own cache. This definition is important, as accepting metadata from the pool may degrade the metadata cache performance for local authentications.

6.5.4 Reducing Interconnect Traffic

Cache pools offer a performance benefit at the cost of increased interconnect traffic. CAPULET proposes four mechanisms to reduce traffic across the interconnect. These mechanisms are important for disaggregated memory systems that implement other protocols that are interconnect bandwidth intensive. When evicting a value from a host's cache in the cache pool, CAPULET broadcasts over the disaggregated memory system to search for an available memory node to store the value. However, doing so requires sending a broadcast, acknowledgment, and data across the interconnect in systems without secure broadcast (systems with secure broadcast only send the broadcast with the data).

The first approach to reduce interconnect traffic uses a probabilistic broadcast mechanism. Local caches only broadcast evicted values with some probability P_e. Doing this reduces the likelihood of remote hits in the cache pool, but also reduces the traffic sent to maintain that data in the pool.

The second approach to reduce interconnect traffic lets remote devices choose to ignore broadcasts of evicted values across the interconnect to store metadata from some remote memory device with probability P_{ie}. This reduces the amount of acknowledgement and data transmission traffic across the interconnect in systems without secure broadcast. Note, in order to ensure that coherence is preserved this means that a host must send an invalidation broadcast when it dirties a block in its local cache, and remote caches cannot ignore invalidation broadcasts.

A third approach to reduce traffic is to let the cache controller choose whether or not to send miss requests to the cache pool with probability P_m. This means that the device will fallback on fetching the metadata from untrusted, local memory even if the value is cached elsewhere in the pool. However, making this decision the local cache reduces the traffic sent across the interconnect.

The fourth approach, much like how remote caches can choose to ignore broadcasts to store evicted values, remote caches can also ignore broadcasts for missed values with probability P_{im}.

6.6 Evaluation

This section describes the evaluation methodology of CAPULET. CAPULET is implemented and evaluated in two formats: ① an extension to CXL-DMSim [289], a timing accurate state-of-the-art simulator extension to gem5 [167] that models a full CXL system, which is used to measure the performance impact of CAPULET and ② a cache emulator coupled with a series of memory traces from SPEC 2017 CPU benchmarks that are used to study the sensitivity of different interleaving strategies on bandwidth and interconnect traffic. The simulator study configures four X86 cores with a 1MB LLC, and a secure memory controller for the local and remote devices with a 64kB metadata cache [109] on the local controller. When using a cache pool, the size of the remote pool is a function of the number of hosts in the system and the access latency is determined by the fabric required to incorporate that many hosts in the system [155]. The trace-based study works from collected memory traces from each of the SPEC CPU 2017 benchmark suite [41] through a gem5 [167] simulation. The region of interest is simulated, as determined by SimPoint [105], and all last-level cache (LLC) misses and writebacks for one trillion instructions are logged. In total, over 800GB of memory trace data were collected. With the collected memory traces, various permutations of accesses are modeled from the traces based on the time of access from the beginning of the region of interest. This approach emulates the "next" memory access to the memory system. The emulation sends each memory access to the emulated memory system by determining to which remote memory the access should be sent based on the address.

The goal of this section is to show the advantages of CAPULET when applied to two popular disaggregated memory system deployments: memory expansion for hosts controlled by end-users (i.e., single program deployments) and for multiple hosts performing arbitrary execution (i.e., multiprogram deployments). The evaluation shows that CAPULET improves performance relative to secure memory on a local device by as much as 4× and 15% on average for HPC workloads. The experiments show a direct correlation between performance benefit of CAPULET and the reduction in metadata accesses for authentication. This benefit is a function of improved metadata cache utilization, which comes at the cost of interconnect traffic.

As described in Section 6.5, CAPULET may be configured with any secure memory implementation so long as the memory system fabric supports the secure sharing of metadata. For ease of implementation, this work implements the protocol described in Gueron's work but the CAPULET technique can be applied to any secure memory implementation that uses metadata caches [99].

6.6.1 Single Program Deployments

This evaluation assumes that there are several hosts using the memory system to achieve a similar end, but hosts are largely idle in their memory system usage. As a result, this section shows the benefit of increased metadata cache capacity despite the longer access latency. This analysis implements CAPULET on top of local and remote metadata caches with a timing associated with local and remote cache access. For simulation purposes, the remote metadata cache is pessimistically configured with the longest latency across the memory fabric. In practice, some hosts will be closer in the interconnect fabric, and communication latency is a function of the distance between these hosts. From here, this work evaluates the benefits of various disaggregated memory configurations with the SPEChpc 2021 benchmarks [156]. The regions of interest are selected based on the existing labeling in each benchmark, following initialization. The analysis is based on the tiny configuration of the SPEChpc workloads to meet the constraints of simulating such a

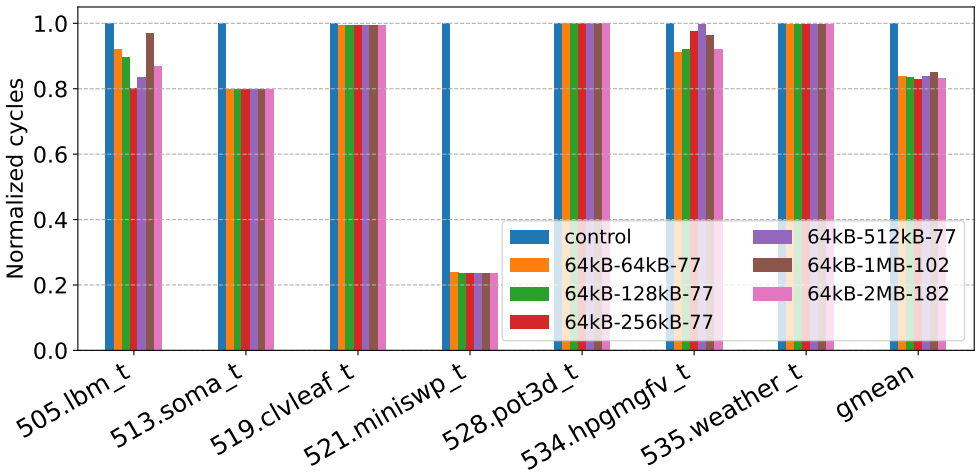


Figure 6.7: Normalized cycles executed across SPEChpc 2021 benchmark suite.

workload, but larger workloads will demonstrate even more memory intensive behavior.

Fig. 6.7 shows the normalized cycles to execute the region of interest for the SPEChpc 2021 workloads (lower is better). The "Control" configuration describes the implementation of the secure memory protocol from prior work on a single local device [99]. Other configurations are specified in the format $lc - rc - rl$ where lc describes the capacity of the local metadata cache in the host of interest's MEE, rc describes the capacity of the remote metadata caches in the "cache pool", and rl describes the end-to-end latency to access remote caches in the cache pool taking into account the disaggregated memory configuration from the memory pooling work [155]. This study pessimistically assumes a system with uniform latency to all devices.

The workloads are run for 1 billion instructions from the beginning of the region of interest, which is a standard approach for simulator-based evaluation [317, 301, 260]. The evaluation shows that CAPULET can benefit performance by up to 4× (`521.miniswp_t`) and by 15-18% on average (`gmean`).

The performance benefits of CAPULET are attributed to the increased "effective" metadata cache capacity that comes from utilizing the space in under-utilized remote metadata caches. As a result, CAPULET reduces the number of memory accesses. The impact of this effect is demonstrated in Fig. 6.8. This figure shows the total number of memory accesses for metadata, normalized to the control. As shown, *the reduction of metadata accesses to the memory device is strongly correlated with the performance benefit.* For instance, the `miniswp` benchmark has approximately 2× fewer metadata memory accesses compared to the control configuration when CAPULET is used in any of the configurations under study. Given that this workload is particularly memory bound, the reduction in additional memory access frees device bandwidth to fetch application data and advance the program state. Similarly, CAPULET reduces memory accesses for metadata fetches by 25% in `soma` which accounts for its additional performance benefit. On the other hand, the `weather` benchmark has a smaller reduction in metadata memory accesses (5% in some configurations) and the corresponding performance benefits are equally small. The reason for this behavior is that this application is mostly compute bound.

Generally, the trends from these workloads demonstrate that optimizations to secure memory are directly related to memory footprint. Consistent with findings in the SPEChpc characterization work, the evaluation

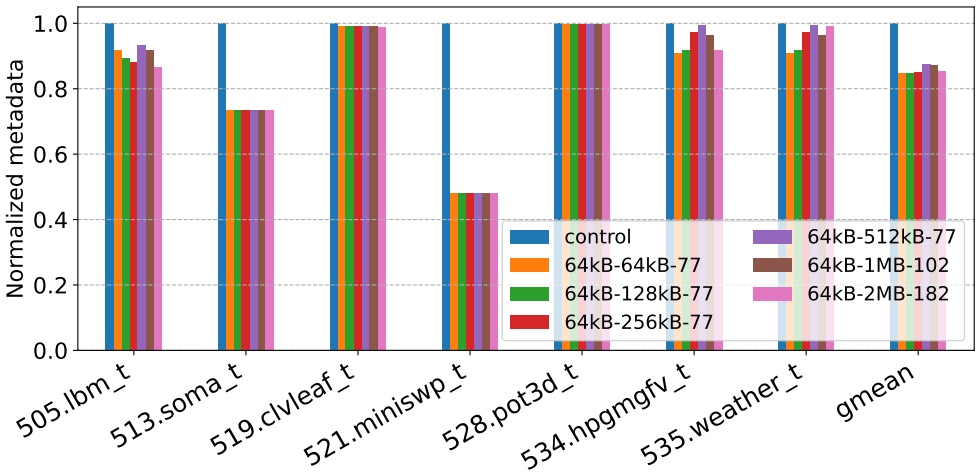


Figure 6.8: Metadata accesses to the memory device. Values are normalized to the number of accesses in the control.

finds that `miniswp` and `soma` are the most memory-bound workloads [40]. That is, they benefit the least from adding compute to the system. Further, the additional capacity of the metadata caches in the cache pool significantly reduced the number of memory accesses due to metadata fetches for each of these workloads. Although the analysis is limited to using the "tiny" inputs by environment constraints, studying these workloads on real hardware shows that the speedup due to extra compute scales even less with small, medium, and large input sizes. Given this observation, it is safe to anticipate that the benefits of CAPULET extend to larger workloads. This is important given the trend of increasing memory footprints in modern applications.

The evaluation emphasizes the benefits of CAPULET by running workloads with configurable size. This is done by using the graph500 [189] breadth-first search (BFS) workload, a random-access microbenchmark, and the partitioned skip list described in Sec. 6.4.2. The graphs are configured to be of sizes 512MB, 1GB, and 2GB (determined using `mperf` [219]), which take 2^{20}, 2^{21}, and 2^{22} nodes as input with 16 edges per node in each, respectively. With this benchmark, the evaluation is based on measurements collected during iterations 8-12 as a representative region of interest and turn off validation. This methodology is consistent with prior work [251]. The random access microbenchmark measures 1 billion random accesses following the initialization of 2GB and 4GB arrays in remote memory. The partitioned skip list prefills 80% an address space of 512K keys, 1M keys, and 2M keys before executing YCSB [61] workload A (50% lookup, 50% updates, random key selection). For each size, the evaluation shows the results of maintaining the lowest 3 levels in remote memory (i.e., 87.5% of the structure).

Figure 6.9 shows cycles executed normalized to the control implementation for each of the size-adjusted workloads. Workloads specified with the "bfs" prefix describe *graph500* workloads, the "mb" prefix refer to the microbenchmark, and the "sl" prefix refers to the partitioned skip list workload where the second value describes the keyspace. The aim of this evaluation is to show that improved metadata cache capacity benefits performance as workloads become more memory intensive (due to their larger memory footprint). This phenomenon is evidenced by the fact that performance improves for bigger memory workloads. CAPULET benefits BFS in a 2GB graph by 18% whereas it only benefits a 512MB graph by 6%. This trend is consistent

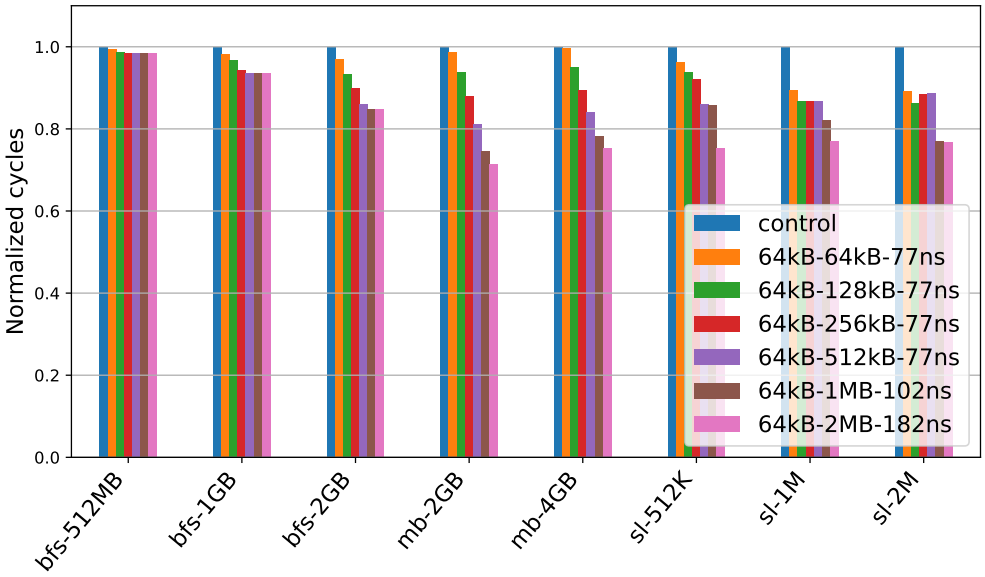


Figure 6.9: Normalized cycles executed for various BFS, random access, and skip list microbenchmark workloads.

for the random access and skip list microbenchmark workloads as well.

6.6.2 Multiprogram Deployments

Beyond local deployments, memory pooling is an anticipated deployment of disaggregated memory in the cloud [155]. Fig. 6.1 shows multiple hosts sharing remote memory devices in the disaggregated memory system to form a memory pool. The protocol is implemented in the operating system and/or interconnect fabric [155, 143, 64, 98]. This evaluation assumes that application data is distributed across remote memory devices at data word granularity by the interconnect fabric to expose peak bandwidth of the memory system. That is, if two hosts share a remote memory device, then its addresses will be split by these hosts.

As discussed in Section 6.6.1, there is a strong correlation between metadata accesses to memory and performance. To emulate program behavior in the cloud, this evaluation is based on multiprogram experiments based on interleaved memory traces from the SPEC CPU 2017 suite [41]. This allows a study of the remote memory system and cache pool behavior for multiple regions of interest in parallel. All accesses to the L3 cache are modeled in the custom emulation model, as well as the protection of all subsequent memory accesses (i.e., misses, writebacks, etc.) with the appropriate secure memory protection. The cache emulator is configured with an L3 cache on each host (two hosts), a secure memory metadata cache and a remote memory device. The emulator is tooled to be able to control interconnect traffic as described in Section 6.5.4. These experiments are performed by toggling the hyperparameters P_e, P_{ie}, P_m, and P_{im}.

Fig. 6.10 shows the impact of applying CAPULET to a shared memory pool in terms of metadata cache utilization and Fig. 6.11 shows the impact on interconnect traffic. Note, each of these analyses are based on two CAPULET configurations: *full* and *partial broadcast*. In the full configuration, P_e, P_{ie}, P_m, and P_{im} are all set to 100%. The partial broadcast column reduces P_e to 20%. Empirically, the cache evicting some metadata or searching for some missed metadata is the primary source of network traffic, so toggling P_{ie} and

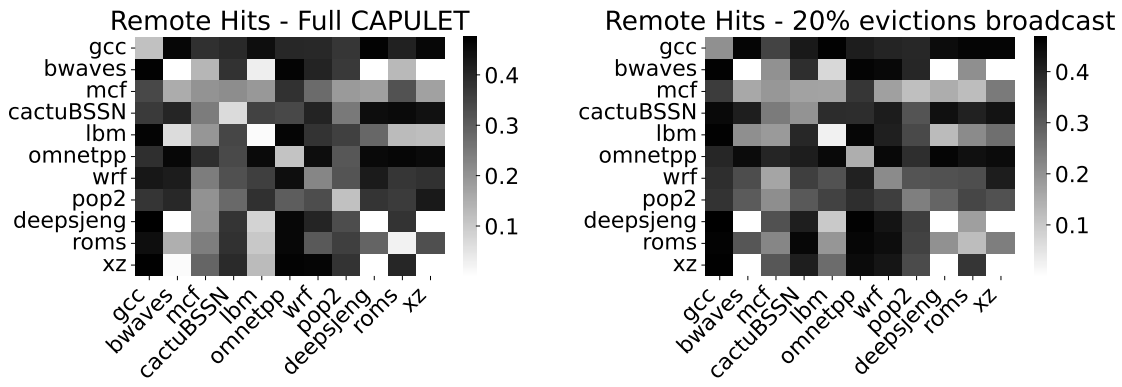


Figure 6.10: Remote hit rates to remote metadata caches (remote hits / local misses) with multiprogram workload.

P_{im} had a negligible impact on hit rates or traffic. That is, if misses are seldom broadcast and/or remote caches ignore evicted data, then the cache pool has no impact on the effective capacity of the local cache. Hit rates (Fig. 6.10) are described as the number of remote hits over the number of local misses; interconnect traffic is described in terms of average messages per metadata cache miss.

Local metadata cache misses can hit elsewhere in the cache pool up to 50% of the time (Fig. 6.10). Along the diagonal (i.e., where the same workload is run on both hosts), the hit rates tend to be lower as both caches in the memory pool will be under similar degrees of strain at the same time. The takeaway of this observation is that, although both applications execute their region of interest, different workloads will go to memory at different rates. Therefore, CAPULET naturally allows any cache in the cache pool to maintain metadata for the application actively making memory demands. The analysis also shows that, perhaps counter intuitively, in some cases remote hit rates tend to improve when reducing P_e. This is attributed to the fact that evicted values are likely to exhibit poor locality, except when the cache experiences significant strain. In these periods it is more likely that values are to be evicted and that they will be found in remote caches.

Fig. 6.11 shows the additional interconnect traffic due to the cache pool in terms of additional messages per data request. The analysis shows that, in the worst case, an additional message may be communicated per data requests. However, note that there is a strong correlation between traffic and remote hit rates. A larger effective capacity in CAPULET is implemented with more interconnect traffic. When CAPULET is less effective, the memory system is not burdened with the negative effect of interconnect traffic overhead due to CAPULET.

6.7 Conclusion

This chapter presents CAPULET, a cache pooling protocol that distributes the load of metadata caches throughout a disaggregated memory system. The cache pool abstraction is powerful; caches can essentially extend their capacity by taking advantage of other devices in the system without adding hardware to the system. Such a primitive is important given that the metadata caches can easily be constrained when tasked

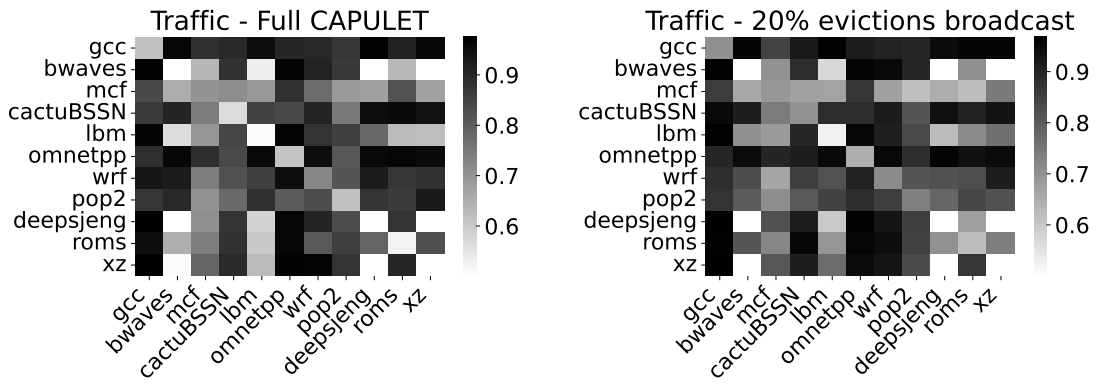


Figure 6.11: Number of additional packets on the interconnect per metadata cache miss with multiprogram workloads.

with caching remote metadata. The properties of disaggregated memories will continue to present new opportunities and challenges for secure memory systems.

There are certain challenges that CAPULET does not address concerning a secure disaggregated memory system. ① The protocol requires that the root of an integrity tree be stored on the host processor, which has no knowledge of the management of the underlying disaggregated memory system. As a result, it is difficult to apply CAPULET to a dynamically sized address space. To do so would require updating the root with the new size, but the root is only securely initialized at the outset of the session. ② Implementing a cache pool requires sending additional traffic across the memory system interconnect under the assumption that sufficient bandwidth exists. If such bandwidth does not exist, the practicality of this primitive is limited.

Chapter 7

Conclusion

This dissertation set out to achieve the goal of improving secure memory to make it more practical for commodity deployment. The works proposed in this dissertation make improvements on the prior literature towards the ends of runtime overhead, spatial overhead, and considers the impact of emerging memory technologies for secure memory.

A challenge of secure memory research is that it is difficult to judge how much a protocol needs to improve the prior literature to be considered "good enough" to become practical. In other words, at what point is there nothing more to say about secure memory? Ultimately, this is a function of two independent problems: ① any security protocol will always incur overhead over an insecure alternative, and ② there is no definition for how slow is "too slow."

Chapter 1 of this dissertation formalizes feature ① in the context of secure memory. The claim made is that secure memory is important because processors make certain implicit assumptions about their memory. The fact that this assumption is implicit gives rise to dangerous execution if it is not held in practice. Therefore, the secure memory protocol *explicitly* implements guarantees around the assumption. However, implementing any guarantee explicitly requires additional work on the critical path of a memory operation. The question of "how much work" must be performed or is reasonable to perform is a function of the robustness of the explicit guarantees. This dissertation makes the argument for one set of guarantees in Chapter 2 given the set of vulnerabilities and attacks that have been demonstrated. Within this context, this dissertation argues that "how much work must be performed" given a set of guarantees to uphold can be reduced by improving the efficiency of the procedure to perform that work. In particular, a general insight that is leveraged in each of the proposed protocols is that the design of the baseline secure memory protocol does not take *usage* of a memory into account. By doing so, a more efficient protocol can reduce the overhead of the protocol required to enforce the guarantees.

A notable limitation of this dissertation is its focus on a single threat model. The challenge of exploring the extent to which this threat model appropriately characterizes the state of the world is an orthogonal problem to the work in this dissertation. Furthermore, the threat model essentially mandates the use of an integrity tree, and Chapter 2 shows that this is the primary performance overhead of secure memory.

Problem ② is much more difficult to quantify, and it is an artifact of any research topic in which performance is a primary concern. In the context of secure memory, the claim that "secure memory is too slow" is strong given that its guarantees have been relaxed in commodity products. If it were fast

enough, there would be no need to relax these guarantees. However, it is far more difficult to make a strong justification that some new proposal is "fast enough." A strong argument for the overhead of a new system or protocol can be made if the approach were implemented and widely deployed. The feedback of end-users would justify if the ends justify the means.

This dissertation makes the claim that the proposed approach in Chapter 3, Chapter 5, and Chapter 6 are "faster than the alternatives" proposed in prior literature. A clear limitation of this dissertation, and any other proposed architectures without commodity deployment, is that this is all that it can claim.

In conclusion, this dissertation satisfies its original goal to make secure memory more practical for commodity deployment. However, this thesis cannot and does not speak to the adjacent question of "at what point will the meaningful things to say about secure memory be exhausted?" nor "how close is the current literature to closing the problem of secure memory?" It is clear that the problem of safely using an untrusted memory is important. It is also clear that prior proposals have been deemed insufficient for commodity deployment. Therefore, it is reasonable to conclude that there is more work to be done towards achieving efficient solutions towards this end. With the problems identified and characterized, this dissertation contributes several proposed extensions to the secure memory protocol and an environment for future work towards achieving a secure memory for commodity deployment.

Appendix A

Secure Memory Simulation Details

The appendix details the design decisions and implementation details of the secure memory implementation in gem5 [167]. The repository is publicly accessible and usable as part of this dissertation[1].

A.1 Design Overview

The simulated secure memory conforms to the gem5's simulation model. The model naturally enables a configuration of various components as well as the ability to add components, such as a secure memory interface. To understand the design methodology of the secure memory simulation, this section first briefly describes the simulation methodology used in gem5.

A.1.1 Nomenclature and gem5 Overview

gem5 [33, 167] is a "timing-accurate" architecture simulator. At a high level, the simulator runs a program as the sequence of hardware events triggered by its execution. Each "event" is represented as a C++ lambda function placed in a priority queue (i.e., the *event queue*) with an associated clock cycle (i.e., a timestamp) at which it should execute. As a result, hardware's behavior is represented by the execution of these lambda functions, where the execution of a hardware event is likely to create another hardware event. Events are associated with hardware *components*, each of which is simulated in the gem5 source.

In order to concretize this model, consider the abstract notion of executing an instruction from a program using gem5's single-stage pipeline processor. To execute the instruction, the processor must fetch the instruction from its memory. The gem5 processor, which maintains the program counter (PC), will begin its execution by placing an event for this fetch in the event queue. This will create a *request* to instruction memory (i.e., the instruction cache, or icache) for the data at the address specified at the PC. The access request and the access itself are not instantaneous, as the request must be received and processed by the icache controller. As a result, the communication between the *ports* connecting the processor to the icache controller may be implemented as a new event or have its timing "accounted" for in some later event (i.e., the tag comparison, access, or response by the cache). That is, part of the lambda function's implementation

[1] `www.github.com/samueltphd/SecureMemory`

Listing A.1: Sample gem5 configuration.

```
import m5
from m5.objects import *

# declare objects
system = System()
system.cpu = X86AtomicSimpleCPU()
system.xbar = SystemXBar()
system.memory = SimpleMemory()

# attach ports
system.cpu.icache_port = system.membus.cpu_side_ports
system.cpu.dcache_port = system.membus.cpu_side_ports
system.membus.mem_side_ports = system.memory.port

# set the workload and run...
```

will entail a communication to another component in which another lambda will be scheduled for some future cycle.

It is worth noting that, within this example, there are several features that highlight the contribution of gem5's abstraction model. ① gem5 merely demands that events are scheduled, but does not dictate the precision at which those events must be described. For instance, most of gem5's "SimpleMemory" schedules events at the granularity of hundreds of cycles (i.e., just the access) whereas DRAMSim [157] based components model finer details such as row refresh. ② The interface of requests and ports naturally allow various components to communicate with one another regardless of the hardware configuration. Ports are classified as generally "requesting" and "receiving" which means that they are component agnostic. In addition, requests are communicated between components via ports in a *packet*. ③ The request *not the packet* serves as the fundamental communication primitive. The processor may issue a request for 8 bytes from memory (e.g., executing the instruction `ld r0, 0(addr)`) which triggers the data cache requesting the full 64 byte cache line from memory. Each of these "events" are associated with the same request but may require different packets to serve the goal of each component.

Given this, gem5 users specify a binary in which the definitions of the possible `SimObject` components are defined and a *configuration* file. The configuration is a Python file that declares the `SimObject`s to use and how they are connected. An example configuration is shown in Listing A.1. The declaration of the `System`, `X86AtomicSimpleCPU`, `SystemXBar`, and `SimpleMemory` refer to objects defined in the simulator source. The interface to attach ports between components is to set one port equal to the other.

More recent version of gem5 explicitly organize components to be called in the configuration front-end. Much like real hardware, these versions leverage a `Board` objects that manages the interaction of processor, cache, and memory components. As demonstrated in Listing A.2, this approach is remains modular enough for a user to specify their desired components. The "memory" component includes the declaration of the memory controller `SimObject` and the memory interface itself (i.e., DDRx, the number of memory channels, etc). Seeing as the work in this dissertation concerns secure memory, which describes an extension of the memory controller logic, implementing a secure memory component in gem5 is exposed to the front-end through a component with the secure memory logic attached to the memory controller.

Listing A.2: Sample gem5 configuration using the `Board` object.

```
import m5
from gem5.isa import ISA
from gem5.components.boards.x86_board import X86Board
from gem5.components.processors.simple_processor import SimpleProcessor
from gem5.components.memory import DualChannel
from gem5.components.cachehierarchies.ruby.mesi_two_level_cache_hierarchy import
    MESITwoLevelCacheHierarchy

# declare objects
board = X86Board(
    processor=SimpleProcessor(isa=ISA.X86, num_cores=2),
    memory=DualChannelDDR4_2400(size="3GiB"),
    cache_hierarchy=MESITwoLevelCacheHierarchy(l1d_size="32KiB", l1i_size="32KiB", l2_size="256
        KiB"),
)

# set the workload and run...
```

A.1.2 Secure Memory Component

This thesis implements secure memory as a component that a gem5 `Board` can incorporate and interact with. In particular, the component intercepts traffic from the `Board`'s last level cache (LLC) before it is sent to memory. The component instantiates the associated secure memory `SimObject`, which is defined in the simulator source. To perform the analysis for this dissertation, the back-end source defines four possible architectures to implement secure memory: ① a direct-encrypted memory (no integrity), ② a counter-mode encrypted memory (no integrity), ③ an encrypted memory (counter-mode) with HMAC authentication, and ④ an encrypted memory (counter-mode) with an integrity tree. Within these configurations, there are a multitude of options to further test various design decisions, which we describe below.

The secure memory component models the timing of the various configurations. This is conveyed in Fig. A.1. In the direct-encryption engine (configuration ①), this includes a "cipher engine" that takes data as input. The input buffer to this engine is served with the plain-text data from write requests and cipher-text data from read requests that have responded from memory. Configurations ②, ③, and ④ all rely on encryption counters as input to the cipher engine to produce a one-time pad (OTP), which is then XOR'ed with the plain-text or cipher-text data for encryption. As a result, the counter must be fetched prior to utilizing the engine (the metadata access is described below). The latency for the cipher engine is configurable, and the engine is assumed to be pipelined (i.e., one request can be processed per cycle). For simplicity, the implementation does not model the actual encryption/decryption of data. Instead, it models the timing that it would take to perform the cipher.

Configurations ③ and ④ rely on a hashing engine so that MACs can authenticate data. To perform the authentication, the untrusted data has its MAC computed (i.e., a hash computation in an HMAC). Therefore, fetched cipher-text data and encryption counter serve as input to the hashing engine to perform an authentication. The latency to perform hashes is configurable, but generally the cipher latency should be longer than the hashing latency. Much like the cipher engine, the implementation only models the timing of the authentication and does not model the actual hashing of data.

Configurations ②, ③, and ④ all rely on metadata. Fig. A.1 shows how metadata may be accessed.

The secure memory component initiates all traffic to the metadata cache and intercepts all traffic from the metadata cache sent to memory. This allows the secure memory implementation to reason about metadata cache hits and misses relative to the larger protocol, which is important to establish roots of trust in configuration ④. To access this metadata in memory without interfering with application data, the memory controller needs to be modified to accept requests outside of the normal address range. Seeing as this secure memory implementation models its timing, the actual storage of each metadata value is unnecessary. Instead, the memory controller is extended to keep a single block where requests for any metadata address are directed.

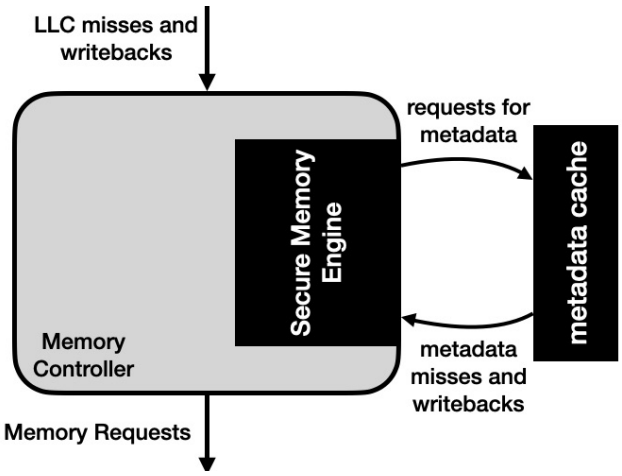


Figure A.1: Model of Secure Memory `SimObject` components.

A.2 Implementation

Beyond the high level description of the components, there are several non-trivial implementation details to optimize the secure memory protocol. These are seldom discussed in the literature, so this section formalizes them.

A.2.1 Parallel Writes and Reads

Consider an outstanding write and read request for the same data or address. Seeing as both require some additional latency on the critical path of the operation for encryption/decryption (regardless of the secure memory configuration), the write request maintains the plain-text data will reside on-chip in a trusted state (it hasn't been sent to memory yet). Therefore, the read request can receive its state from the pending write rather than receiving the data from memory. In doing so, the read does not require a memory access nor the cipher.

When handling a read request, the secure memory component will check for a parallel write at two points: before the memory fetch and after the memory response. To do so, it checks the input buffer to the cipher engine for parallel writes. Seeing as the memory authentication may require significant additional latency on the critical path of the read, write requests also check the active read requests that are awaiting authentication. If a corresponding request exists, the authentication can be bypassed as the data fetched from memory is stale relative to the application state. Note, this may slightly modify the memory consistency model of the memory system, but the modification will only decrease how relaxed the consistency model will appear. Otherwise, this protocol is required in order to implement strict consistency in a secure memory.

A.2.2 Concurrent Authentications

To implement concurrent authentications that share some metadata, the secure memory component needs to have some means for requests to track their associated children. Given the presence of the metadata cache, the authentication data may return before or after its children. As such, the secure memory component maintains a buffer for children awaiting an authenticating parent as well as for parents awaiting requested children to respond.

In either case, there may be multiple parallel requests or a incoming future request may arrive that shares an existing parent value. For this reason, the first step to authenticate some request is to announce that it is now awaiting authentication. Any request for some authenticating metadata is first searched for in the pending structures (either requests or responses) before a new request is made. If the parent metadata has already been requested, then it will authenticate this child whenever it is found. The structures to track parent requests must maintain a counter to ensure that all children depending on the metadata for authentication are served before the value is discarded. Note, a node may be both a parent node *and* a child node if the metadata is untrusted, and a node serves the role of child before it acts as a parent.

A.2.3 Concurrent Updates

In this implementation, the secure memory component heavily relies on the metadata cache to coordinate parallel updates to the metadata state. Practically, the component will pessimistically send all update requests to the security metadata to the metadata cache. Seeing as the cache acts as a *write allocate* cache (i.e., misses are fetched to fill the associated cache block before the block is written), parallel writes for the same block will be concurrently handled by the cache. That is, the cache will allocate an MSHR register for each of the writes and these writes will all be serviced in the order that they arrive when the block responds from memory. Theoretically, if n writes of the same block (i.e., tree node) occur in parallel, only write $n-1$ needs to be performed to appropriately update the tree node. To alleviate some of this pressure, the secure memory component could implement logic in the port to detect these parallel writes and reduce the metadata cache traffic, but for simplicity this implementation offloads that task to the cache.

A.2.4 Responding to the Processor

Seeing as decryption and authentication are asynchronous operations whose latency are dependent on cache behaviors, the secure memory component needs to track when each of these procedures have completed for a particular request. As such, the implementation maintains two buffers for authenticated and decrypted read requests from memory. In addition, the component maintains a lightweight engine that sends responses with the data back to the processor once it appears in both buffers.

A.3 Tutorial

Using the secure memory component is designed to be equivalent to using the gem5 simulator with a its associated memory component. As such, the "user guide" for secure memory with gem5 is consistent with the rest of the simulator. This is well documented[2]. To use a secure memory component, a user can import

[2] `https://www.gem5.org/documentation` and `https://www.gem5.org/documentation/learning_gem5/introduction`.

Listing A.3: Sample gem5 configuration using the `Board` object and a secure memory component.

```
import m5
from gem5.isa import ISA
from gem5.components.boards.x86_board import X86Board
from gem5.components.processors.simple_processor import SimpleProcessor
from gem5.components.memory.secure import (
    DirectEncryptedMemory,
    CounterModeEncryptedMemory,
    MacAuthenticatedMemory,
    IntegrityTreeAuthenticatedMemory,
)
from gem5.components.cachehierarchies.ruby.mesi_two_level_cache_hierarchy import
    MESITwoLevelCacheHierarchy

# declare objects
board = X86Board(
    processor=SimpleProcessor(isa=ISA.X86, num_cores=2),
    memory=SecureMemory(size="3GiB"),
    cache_hierarchy=MESITwoLevelCacheHierarchy(l1d_size="32KiB", l1i_size="32KiB", l2_size="256
        KiB"),
)

# set the workload and run...
```

one of the components implemented in this work into their front-end configuration. An example of this is in the Listing A.3.

Note, the provided implementation includes several possible memory controller extensions for various secure memory implementations. These are `DirectEncryptedMemory`, `CounterModeEncryptedMemory`, `MacAuthenticatedMemory`, `IntegrityTreeAuthenticatedMemory`, and each configures and implements an associated `SimObject` in the simulator back-end. Within each of these configurations, there are several potential options that can be passed to the constructor. These are enumerated in the `help` flags, and describe various potential configurations of the design decisions in the secure memory implementation.

This section also describes the methodology to extend the infrastructure and provides some insight into common errors that may arise.

A.3.1 Extending the Secure Memory Component

It is impossible to foresee all of the possible proposed modifications to the secure memory component. Therefore, the implementation details of the works described in this dissertation serve as case studies for these changes.

Cordelia In order to implement Cordelia, the integrity tree structure must be modified to become pointer-chasing. To account for this, the memory controller and abstract memory interface `SimObject`s need to be modified to allocate the memory for the integrity tree and initialize the pointers for a balanced tree on simulator boot. This can be achieved by overriding the objects `startup` function. In addition, requests for this metadata must be appropriately pointed towards these addresses in the memory controller.

After modifying the memory controller and abstract memory interfaces to make the space for the integrity tree metadata, the secure memory component needs to be modified such that parent addresses are no longer computed based on the child address. Instead, they require the response from the memory fetch and the

metadata state is read to know the parent value. Seeing as the secure memory component controls traffic to and from the metadata cache, this read does not need to happen before the response from the metadata cache, but it cannot be as eager as a computation on the current address. This organization also allows the secure memory component to write to metadata, which restructures the tree shape. Cordelia leverages this to restructure the tree shape according to a Huffman tree update scheme.

Baobab Merkle Tree It is necessary to maintain the true state of the integrity tree leaves in order to implement the Baobab Merkle Tree. To maintain leaves, space needs to be allocated in the memory components in the same way as the integrity tree nodes in Cordelia. Space needs to be allocated in the secure memory component to maintain the memoization table. Much like updating pointers in Cordelia, the indices in the leaves are updated by writing to their state.

A Midsummer Night's Tree To implement the AMNT architecture, it is necessary to ensure that the secure memory component implements a crash consistent metadata update scheme. This entails setting the `WRITE_THROUGH` flag associated with the packet to ensure that the changes are propagated downstream from the cache for the appropriate packets. When implementing leaf persistence, this is only done for packets at the encryption counter level. For strict persistence, this entails all packets.

AMNT implements leaf persistence in a fast subtree. This is implemented by tracking an additional address field in the secure memory component associated with an integrity tree node. From here, a check is performed prior to creating a sequence of writes to determine if the current update path occurs inside or outside of the subtree and uses the correct crash consistency protocol accordingly. In addition, the authentication procedure is updated such that reaching the subtree root is trusted in the same way as the true root or a metadata cache hit.

CAPULET CAPULET does not modify anything about the secure memory protocol in and of itself. Instead, seeing as CXL is not implemented in gem5, the implementation includes multiple memory controllers and misses are used to coordinate among one another. The biggest implementation detail is to model the latency of a remote cache communication, which is implemented as a scheduled event in which the components check each other's state to handle the request.

A.3.2 Common Errors

Port Communication The `recvTimingReq` and `recvTimingResp` functions associated with a port return a boolean value. Without loss of generality, consider an incoming request. If the `SimObject` cannot handle the incoming request, then the `recvTimingReq` function will return `false`. If it does so, then the port on the other side of the connection will maintain the packet and await some later notification that the `SimObject` has the capacity to now handle that request. It does so by calling the `sendRetryReq` function. The interaction between ports is not unique to the secure memory component, but these errors are common as the secure memory component is responsible for managing ports to several entities.

Using the Metadata Cache Whenever a request is made to a cache component, the implementation of a cache `SimObject` expects a response. That is, if the `metadata_response_port` receives a request from

the metadata cache (i.e., a miss, coherence request, etc), then the cache block is blocked until a response is returned. As such, it is important that all requests have an equivalent response.

Tracking Packets versus Requests It is worth distinguishing the roles of a `Packet` versus a `Request` in gem5. `Packet`s carry requests between various `SimObject`s, but if `SimObject` A communicates `Packet` p with `Request` r to `SimObject` B, then it is not a guarantee that p will be used if B later sends r to `SimObject` C. As such, if a component wants to track the behavior of an outstanding request, it is important to track the `Request` object in the event that the packet is freed, especially if its state will be later examined. Alternatively, the addresses can be tracked to avoid a reference to the `Request` object in a structure in the object.

Deadlock If the `Board` class implements a `ruby` based cache hierarchy, an external monitor tracks the state of active requests. If they are not processed in a reasonable amount of time, then the monitor will throw a "deadlock" error. These errors occur when the processor state is expecting a response from the memory system for some request that it has not received. When such an error occurs, it is worth examining the state of the secure memory component to determine which requests are currently pending and the state of the metadata.

Dynamic Memory When creating a packet, the memory for data is not allocated by default. If an error is thrown when attempting to copy data from memory to a metadata request (read) or vice versa (write), then it is worth verifying that the space for that data was allocated (i.e., calling `pkt->allocate()`). By the same token, it is important to free the packet pointer whenever the secure memory component no longer requires its state.

Appendix B

Compression Proposal

B.1 Motivation

To implement secure memory, the memory controller must maintain some amount of metadata in memory alongside the data to authenticate its state. As described in Chapter 4, the storage requirement of this metadata is a significant proportion of the capacity. While Chapter 4 proposes a scheme in which the spatial overhead of the BMT can be reduced by memoizing encryption counters, it faces several limitations.

Limitation 1. For one, the memoization table requires significant on-chip area overhead for its component. On-chip space is limited, and reserving a 256kB table in addition to the metadata cache exceeds the capacity allocated to the memory controller in SGX [109].
Goal: Do not require further on-chip area to reduce storage overhead.

Limitation 2. In addition, the proposal from Chapter 4 describes storing additional metadata in memory (i.e., the reverse mapping table) to reduce the storage overhead of other metadata. The notion of adding in-memory storage to reduce the global memory storage overhead is unsatisfactory; it limits the applicability further advances of the alternative baseline.
Goal: Do not require additional metadata fields to be stored in memory alongside the data and secure memory metadata.

Limitation 3. The proposal from Chapter 4 reduces the spatial overhead of the encryption counters and the integrity tree. However, as highlighted in Sec. 4.2, these components make up a small percentage of the overall storage overhead of secure memory metadata.
Goal: Reduce the storage overhead of all fields, *including* the MACs.

As such, this dissertation proposes a co-designed secure and compressed memory architecture. In such a proposal, data is compressed so that the "effective" capacity of the memory device is increased. In fact, prior work has shown that memory compression can improve the effective size of the device by $2-4\times$ in practice [17, 16]. Then, the secure memory metadata can protect the raw bytes stored in the device itself

rather than the compressed data. As such, the storage overhead of secure memory metadata is reduced by the ratio at which the data can be compressed. There are a few insights of such a proposal:

1. The storage overhead of the secure memory metadata becomes a function of the efficacy of the compressed memory system. As a result, the security metadata can take the same form as in a decompressed scheme, and the security primitives are upheld (e.g., hashes are equally unlikely to collide, etc).

2. The cost of compression can be hidden behind the cost of maintaining the secure memory metadata. Thus, the property of compression comes for "free."

3. This proposal argues in favor of co-designing security features with other desirable features of a system or architecture. Much of the literature in developing efficient security protocols harp on performance. Alternatively, this proposal takes the approach that secure systems/architectures may be more palatable with additional features.

B.2 Challenges and Related Work

There are several challenges to implement the proposed co-designed secure-compressed memory architecture. This section describes some of the key challenges around implementing such a scheme. In addition, this section overviews some of the prior literature on compressed memory, which speaks to some of these challenges.

B.2.1 Challenges

Challenge 1. There is a wide breadth of compression algorithms [185] that can potentially be used. Each has its own properties with respect to latency, complexity, size stability of compressed data, etc. For instance, Lempel-Ziv (LZ or LZ77) [315] compression provides high compressibility of data, but is considered a slow operation with relatively limited opportunity for parallelization. Other algorithms, such zero-content detection, are much faster but whose compressibility is more heavily dependent on the prevalence of patterns in the data.

Challenge 2. Once the data is compressed, it is non-trivial to know where the data is stored. Suppose the compressed data is stored at the physical address specified by the processor. In such a case, the compressed data uses a small percentage of the word in memory, and the unused bytes in the word are unreachable. That is, this scheme suffers from internal fragmentation. Thus, some other scheme to manage a compressed memory must be used.

Challenge 3. The capacity of a compressed memory becomes a function of the data contents. At the same time, the operating system (OS) and other software systems rely on knowing the memory size a priori to maintain and processes. For example, a process that uses a greater memory size than the device capacity may be killed by the OS if the contents of its memory change, even if the process does not allocate more memory. As a result, the OS must be aware of the compressed memory scheme.

B.2.2 Related Work

The notion of a compressed memory has been discussed for several decades [148, 104]. Mittal and Vetter [185] identify data compressibility, compression granularity, choice of compression algorithm, latency and energy overhead, and implementation complexity as the key issues in a compressed memory. In the context of this proposal, the granularity of data is the data word and/or cache line size (i.e., 64-bytes). In addition, the choice of compression algorithm is unrestricted insofar as the latency can be hidden by the AES latency (i.e., 53 cycles).

Regarding Challenge 1, IBM's memory expansion technology (MXT) [264] indicates that compression latency can successfully be hidden behind the encryption latency. MXT leverages LZ77 compression algorithm as an ASIC, and claims that 16 bytes can be compressed per cycle. This indicates that even data words of 512 bytes can be compressed in less latency than the latency to produce the OTP. As such, the compression algorithm may be aggressive, but algorithms that are difficult to implement on an ASIC (i.e., Huffman encoding [187] relies on additional metadata to access) should not be considered.

To account for Challenge 2, the MXT architecture proposes using an on-chip memory-resident table to perform "address indirection." This approach distinguishes the notion of data's physical address from the processor's physical address (i.e., the *logical physical address*, or LPA). When accessing some location in memory, the memory controller translates the requested logical physical address to the true physical address to be accessed in the device. This allows the memory controller to remap data as its compressibility changes whenever the data is to be stored. Alternatively, Ekman and Stenstrom [78] propose a scheme in which translation can reside in an in-memory structure much like the TLB. Alternative approaches aim to keep compression cost low by keeping a compressed and uncompressed region that dynamically detect the working sets of memory [144, 195]. This would not be an issue in this proposed work, as the compression latency is not a limiting feature.

Challenge 3 is largely out of scope of the proposed solution. Practical systems use interfaces such as `zswap` [129] or allowing software to manage logical partitions that are mapped onto the physical memory space [111]. The task is also similar to how an OS maintains a remote address space in a far memory system [144].

To summarize, the prior work on compressed memory indicates that an aggressive compression algorithm may be used, as even the latency to compress data using LZ77 can be hidden behind the latency to produce a OTP. Furthermore, the memory controller will need to maintain some mapping of logical physical addresses of the decompressed data to the physical addresses of the compressed data in the memory device. The task of implementing software to use the architecture is beyond the scope of the proposed architecture, but the area of designing systems to leverage these architectures is well studied.

B.3 Design

B.3.1 Pipeline

The compression methodology for a compressed memory is non-trivial, especially in the context of a secure memory. In particular, there are implications if the data is compressed *before* or *after* it is encrypted. If data is encrypted before it is compressed, then it is unlikely that the encrypted data will have enough patterns

to be compressible. On the other hand, if data is encrypted after it is compressed, then a smaller portion of the one-time pad (OTP) used for encryption will be utilized in the cipher.

This proposed design considers using the pipeline depicted in Fig. B.1. In order to know which metadata to use, the compression must happen before encryption and metadata updates. In the pipeline, the first stage is to compress the data. Seeing as the new data contents may impact the data's compressibility, the data may need to be remapped as the new compressed data may no longer fit in its prior physical address. The next stage of the pipeline is to remap the compressed data. The mapping procedure is detailed in Sec. B.3.2, and the production of a new physical address will indicate which metadata is associated with the data. As such, the final stage of the pipeline refers to the maintenance of the metadata fields. This includes producing the OTP, encrypting the compressed data, producing the MAC, and updating the integrity tree.

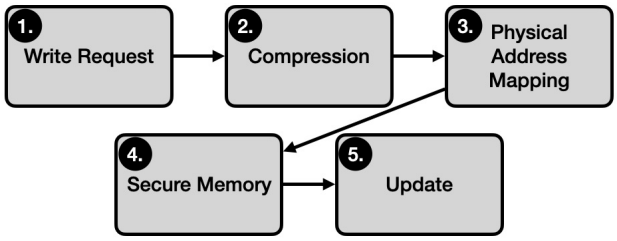


Figure B.1: Pipeline of stages to compress a secure memory.

There are a few things to note about the proposed pipeline. For one, the compression stage does not make any assumptions about the compression algorithm used. Seeing as all data writes must go through this stage, it may be desirable to use a pipelined, low-latency engine. It is important that the engine uses a safe, sharable compression algorithm to be resilient to data leakage via BREACH, DROWN, CRIME, and BEAST attacks [125]. In these attacks, an adversary sends payloads to memory whose compressibility is known. The produced MACs leak information about the other victim data stored in the shared word.

Furthermore, the remapping of newly compressed data will require re-encrypting and recomputing the MAC of the word in which the data is to be stored. This means that data writes in the secure-compressed memory also entails fetching the word at which this data is mapped.

B.3.2 Mapping Logical Physical Addresses

The proposed architecture must map logical physical addresses (LPAs) to physical addresses (PAs) in the memory device. The structures to implement this are depicted in Fig. B.2. This is achieved using a look-up table (which can reside in memory) that maintains the LPA, the PA, and the size of the compressed data. In addition, the architecture must maintain a set "free list" of unmapped addresses of various sized "chunks." For simplicity, the candidate chunk sizes are 16 bytes, 32 bytes, and 64 bytes.

Look-Up Table

Logical Physical Address	Physical Address	Size
L_A	P_A	64
L_B	P_B	16
L_C	P_C	32

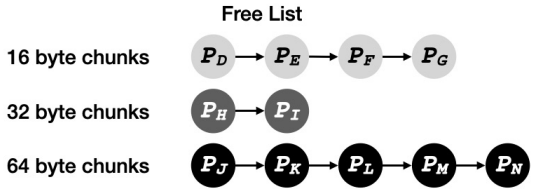


Figure B.2: Data structures required to perform mapping of logical physical addresses to physical addresses in the memory device. Includes a look-up table for actively mapped addresses and a free list for unmapped addresses.

Data that can be compressed to 16 bytes or less can be mapped to the 16 byte chunks, data that can be compressed to 32 bytes or less (but more than 16 bytes) are mapped to a 32 byte chunk. Data that cannot be compressed to 32 bytes or less is kept decompressed and mapped to a 64 byte chunk. Whenever data is remapped, the PA in the prior entry of the look-up table is added to the "free list" with the associated size. The "free list" may be maintained much like the buddy allocation system to ensure that an appropriate sized chunk always exists. This problem is explored in detail in MXT [264].

Bibliography

[1] *4GB (x64, SR) 260-Pin DDR4 SODIMM*. MTA4ATF51264HZ. Micron. Aug. 2023.

[2] Rahaf Abdullah, Huiyang Zhou, and Amro Awad. "Plutus: Bandwidth-efficient memory security for GPUs". In: *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2023, pp. 543–555.

[3] Rahaf Abdullah et al. "Salus: Efficient Security Support for CXL-Expanded GPU Memory". In: *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2024, pp. 1–15.

[4] Shaizeen Aga and Satish Narayanasamy. "Invisimem: Smart memory defenses for memory bus side channel". In: *ACM SIGARCH Computer Architecture News* 45.2 (2017), pp. 94–106.

[5] Minseon Ahn et al. "Enabling CXL memory expansion for in-memory database management systems". In: *Proceedings of the 18th International Workshop on Data Management on New Hardware*. 2022, pp. 1–5.

[6] Mohammad Alshboul et al. "Bbb: Simplifying persistent programming using battery-backed buffers". In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2021, pp. 111–124.

[7] Mazen Alwadi and Amro Awad. "Caching techniques for security metadata in integrity-protected fabric-attached memories". In: *EAI Endorsed Transactions on Security and Safety* 7.24 (2020).

[8] Mazen Alwadi, Aziz Mohaisen, and Amro Awad. "Phoenix: Towards persistently secure, recoverable, and nvm friendly tree of counters". In: *arXiv preprint arXiv:1911.01922* (2019).

[9] Mazen Alwadi, Aziz Mohaisen, and Amro Awad. "Promt: optimizing integrity tree updates for write-intensive pages in secure nvms". In: *Proceedings of the ACM International Conference on Supercomputing*. 2021, pp. 479–490.

[10] Mazen Alwadi et al. "Minerva: Rethinking Secure Architectures for the Era of Fabric-Attached Memory Architectures". In: *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2022, pp. 258–268.

[11] Emmanuel Amaro et al. "Logical Memory Pools: Flexible and Local Disaggregated Memory". In: *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*. 2023, pp. 25–32.

[12] Yuda An et al. "A Novel Extensible Simulation Framework for CXL-Enabled Systems". In: *arXiv preprint arXiv:2411.08312* (2024).

[13] Ittai Anati et al. “Intel Software Guard Extensions (Intel SGX)”. In: *Tutorial at International Symposium on Computer Architecture (ISCA)*. 2015.

[14] *Apple Platform Security*. `https://help.apple.com/pdf/security/en_US/apple-platform-security-guide.pdf`. Accessed: 2025-02-23.

[15] Andrea Arcangeli, Izik Eidus, and Chris Wright. “Increasing memory density by using KSM”. In: *Proceedings of the linux symposium*. Citeseer. 2009, pp. 19–28.

[16] Angelos Arelakis and Per Stenstrom. “SC2: A statistical compression cache scheme”. In: *ACM SIGARCH Computer Architecture News* 42.3 (2014), pp. 145–156.

[17] Angelos Arelakis and Per Stenström. “A case for a value-aware cache”. In: *IEEE Computer Architecture Letters* 13.1 (2012), pp. 1–4.

[18] Zaid Al-Ars et al. “DRAM-specific space of memory tests”. In: *2006 IEEE International Test Conference*. IEEE. 2006, pp. 1–10.

[19] Gal Assa, Michal Friedman, and Ori Lahav. “A Programming Model for Disaggregated Memory over CXL”. In: *arXiv preprint arXiv:2407.16300* (2024).

[20] Hagit Attiya et al. “Detectable recovery of lock-free data structures”. In: *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2022, pp. 262–277.

[21] Damien Aumaitre and Christophe Devine. “Subverting windows 7 x64 kernel with dma attacks”. In: *HITBSecConf Amsterdam* (2010).

[22] Hillel Avni, Nir Shavit, and Adi Suissa. “Leaplist: lessons learned in designing tm-supported range queries”. In: *Proceedings of the 2013 ACM symposium on Principles of distributed computing*. 2013, pp. 299–308.

[23] Amro Awad et al. “Persistently-secure processors: Challenges and opportunities for securing non-volatile memories”. In: *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2019, pp. 610–614.

[24] Amro Awad et al. “Triad-nvm: Persistency for integrity-protected and encrypted non-volatile memories”. In: *Proceedings of the 46th International Symposium on Computer Architecture*. 2019, pp. 104–115.

[25] Andrew Baumann, Marcus Peinado, and Galen Hunt. “Shielding applications from an untrusted cloud with haven”. In: *ACM Transactions on Computer Systems (TOCS)* 33.3 (2015), pp. 1–26.

[26] Scott Beamer, Krste Asanović, and David Patterson. “The GAP benchmark suite”. In: *arXiv preprint arXiv:1508.03619* (2015).

[27] Michael Becher, Maximillian Dornseif, and Christian N Klein. “FireWire: all your memory are belong to us”. In: *Proceedings of CanSecWest* 67 (2005).

[28] Laszlo A. Belady. “A study of replacement algorithms for a virtual-storage computer”. In: *IBM Systems journal* 5.2 (1966), pp. 78–101.

[29] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. “Keying hash functions for message authentication”. In: *Advances in Cryptology—CRYPTO’96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*. Springer. 1996, pp. 1–15.

[30] Mihir Bellare et al. “A concrete security treatment of symmetric encryption”. In: *Proceedings 38th Annual Symposium on Foundations of Computer Science*. IEEE. 1997, pp. 394–403.

[31] Daniel S Berger et al. “Design tradeoffs in CXL-based memory pools for public cloud platforms”. In: *IEEE Micro* 43.2 (2023), pp. 30–38.

[32] Christian Bienia et al. “The PARSEC benchmark suite: Characterization and architectural implications”. In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 2008, pp. 72–81.

[33] Nathan Binkert et al. “The gem5 simulator”. In: *ACM SIGARCH computer architecture news* 39.2 (2011), pp. 1–7.

[34] Manuel Blum et al. “Checking the correctness of memories”. In: *Algorithmica* 12 (1994), pp. 225–244.

[35] Daniel G Bobrow et al. “TENEX, a paged time sharing system for the PDP-10”. In: *Communications of the ACM* 15.3 (1972), pp. 135–143.

[36] David Boles, Daniel Waddington, and David A Roberts. “CXL-enabled enhanced memory functions”. In: *IEEE Micro* 43.2 (2023), pp. 58–65.

[37] William Bolosky, Robert Fitzgerald, and Michael Scott. “Simple but effective techniques for NUMA memory management”. In: *Proceedings of the twelfth ACM symposium on Operating Systems Principles*. 1989, pp. 19–31.

[38] Rodrigo Branco and Shay Gueron. “Blinded random corruption attacks”. In: *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE. 2016, pp. 85–90.

[39] Rory Breuk and Albert Spruyt. “Integrating DMA attacks in exploitation frameworks”. In: *University of Amsterdam, Tech. Rep* (2012), pp. 2011–2012.

[40] Holger Brunst et al. “First experiences in performance benchmarking with the new SPEChpc 2021 suites”. In: *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE. 2022, pp. 675–684.

[41] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. “SPEC CPU2017: Next-generation compute benchmark”. In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 2018, pp. 41–42.

[42] Irina Calciu et al. “Black-box concurrent data structures for NUMA architectures”. In: *ACM SIGPLAN Notices* 52.4 (2017), pp. 207–221.

[43] Irina Calciu et al. “Rethinking software runtimes for disaggregated memory”. In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2021, pp. 79–92.

[44] Brian D Carrier and Joe Grand. “A hardware-based memory acquisition procedure for digital investigations”. In: *Digital Investigation* 1.1 (2004), pp. 50–60.

[45] John Carter et al. “Impulse: Building a smarter memory controller”. In: *Proceedings Fifth International Symposium on High-Performance Computer Architecture*. IEEE. 1999, pp. 70–79.

[46] Miguel Castro, Barbara Liskov, et al. “Practical byzantine fault tolerance”. In: *OsDI*. Vol. 99. 1999. 1999, pp. 173–186.

[47] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. "Atlas: Leveraging locks for non-volatile memory consistency". In: *ACM SIGPLAN Notices* 49.10 (2014), pp. 433–452.

[48] Mango C-T Chao et al. "Fault models for embedded-DRAM macros". In: *Proceedings of the 46th Annual Design Automation Conference*. 2009, pp. 714–719.

[49] Yangyin Chen. "ReRAM: History, status, and future". In: *IEEE Transactions on Electron Devices* 67.4 (2020), pp. 1420–1433.

[50] Zhengguo Chen, Youtao Zhang, and Nong Xiao. "Cachetree: Reducing integrity verification overhead of secure nonvolatile memories". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.7 (2020), pp. 1340–1353.

[51] Zhengguo Chen, Youtao Zhang, and Nong Xiao. "ExtraCC: Improving Performance of Secure NVM with Extra Counters and ECC". In: *The 36th International Conference on Massive Storage Systems and Technology*. 2020.

[52] Siddhartha Chhabra and Yan Solihin. "i-NVMM: A secure non-volatile main memory system with incremental encryption". In: *International Symposium on Computer Architecture*. 2011, pp. 177–188.

[53] Siddhartha Chhabra et al. "SecureME: a hardware-software approach to full system security". In: *Proceedings of the international conference on Supercomputing*. 2011, pp. 108–119.

[54] Ping Chi et al. "Architecture design with STT-RAM: Opportunities and challenges". In: *2016 21st Asia and South Pacific design automation conference (ASP-DAC)*. IEEE. 2016, pp. 109–114.

[55] Pierre Chor-Fung Chia, Shi-Jie Wen, and Sang H Baeg. "New DRAM HCI qualification method emphasizing on repeated memory access". In: *2010 IEEE International Integrated Reliability Workshop Final Report*. IEEE. 2010, pp. 142–144.

[56] Kwanghoon Choi et al. "ShieldCXL: A Practical Obliviousness Support with Sealed CXL Memory". In: *ACM Transactions on Architecture and Code Optimization* (2024).

[57] Md Hafizul Islam Chowdhuryy and Fan Yao. "IvLeague: Side Channel-Resistant Secure Architectures Using Isolated Domains of Dynamic Integrity Trees". In: *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2024, pp. 1153–1168.

[58] Dwaine Clarke et al. "Incremental multiset hash functions and their application to memory integrity checking". In: *Advances in Cryptology-ASIACRYPT 2003: 9th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, November 30–December 4, 2003. Proceedings 9*. Springer. 2003, pp. 188–207.

[59] Dwaine Clarke et al. "Towards constant bandwidth overhead integrity checking of untrusted data". In: *2005 IEEE Symposium on Security and Privacy (S&P'05)*. IEEE. 2005, pp. 139–153.

[60] Joel Coburn et al. "NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories". In: *ACM SIGARCH Computer Architecture News* 39.1 (2011), pp. 105–118.

[61] Brian F Cooper et al. "Benchmarking cloud serving systems with YCSB". In: *Proceedings of the 1st ACM symposium on Cloud computing*. 2010, pp. 143–154.

[62] Victor Costan and Srinivas Devadas. "Intel SGX explained". In: *Cryptology ePrint Archive* (2016).

[63] Crispin Cowan et al. “Buffer overflows: Attacks and defenses for the vulnerability of the decade”. In: *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*. Vol. 2. IEEE. 2000, pp. 119–129.

[64] Heather Craddock et al. “The Case for Physical Memory Pools: A Vision Paper”. In: *Cloud Computing–CLOUD 2019: 12th International Conference, Held as Part of the Services Conference Federation, SCF 2019, San Diego, CA, USA, June 25–30, 2019, Proceedings 12*. Springer. 2019, pp. 208–221.

[65] Tyler Crain, Vincent Gramoli, and Michel Raynal. “A contention-friendly, non-blocking skip list”. PhD thesis. INRIA, 2012.

[66] Tyler Crain, Vincent Gramoli, and Michel Raynal. “No hot spot non-blocking skip list”. In: *2013 IEEE 33rd International Conference on Distributed Computing Systems*. IEEE. 2013, pp. 196–205.

[67] *CXL Specification*. `https://computeexpresslink.org/cxl-specification/`. Accessed: 19-07-2024.

[68] Henry Daly et al. “NUMASK: high performance scalable skip list for NUMA”. In: *32nd International Symposium on Distributed Computing (DISC 2018)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. 2018, pp. 18–1.

[69] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. “An Introduction to the Compute Express Link (CXL) Interconnect”. In: *ACM Computing Surveys* 56.11 (2024), pp. 1–37.

[70] Andrea Di Dio et al. “Copy-on-Flip: Hardening ECC Memory Against Rowhammer Attacks.” In: *NDSS*. 2023.

[71] Nan Ding et al. “Evaluating the potential of disaggregated memory systems for HPC applications”. In: *Concurrency and Computation: Practice and Experience* (2023), e8147.

[72] Chunfeng Du et al. “ESD: An ECC-assisted and Selective Deduplication for Encrypted Non-Volatile Main Memory”. In: *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2023, pp. 977–990.

[73] Chunfeng Du et al. “FSDedup: Feature-Aware and Selective Deduplication for Improving Performance of Encrypted Non-Volatile Main Memory”. In: *ACM Transactions on Storage* (2024).

[74] Zhuohui Duan et al. “Gengar: an RDMA-based distributed hybrid memory pool”. In: *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2021, pp. 92–103.

[75] Guillaume Duc and Ronan Keryell. “Cryptopage: An efficient secure architecture with memory encryption, integrity and information leakage protection”. In: *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE. 2006, pp. 483–492.

[76] Loïc Duflot, Yves-Alexis Perez, and Benjamin Morin. “What if you can't trust your network card?” In: *Recent Advances in Intrusion Detection: 14th International Symposium, RAID 2011, Menlo Park, CA, USA, September 20-21, 2011. Proceedings 14*. Springer. 2011, pp. 378–397.

[77] Morris J Dworkin. *Sp 800-38c. recommendation for block cipher modes of operation: The ccm mode for authentication and confidentiality*. 2004.

[78] Magnus Ekman and Per Stenstrom. “A robust main-memory compression scheme”. In: *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE. 2005, pp. 74–85.

[79] Reouven Elbaz et al. “A parallelized way to provide data encryption and integrity checking on a processor-memory bus”. In: *Proceedings of the 43rd annual Design Automation Conference*. 2006, pp. 506–509.

[80] Reouven Elbaz et al. “Hardware mechanisms for memory authentication: A survey of existing techniques and engines”. In: *Transactions on Computational Science IV: Special Issue on Security in Computing* (2009), pp. 1–22.

[81] Reouven Elbaz et al. “Tec-tree: A low-cost, parallelizable tree for efficient defense against memory replay attacks”. In: *Cryptographic Hardware and Embedded Systems-CHES 2007: 9th International Workshop, Vienna, Austria, September 10-13, 2007. Proceedings 9*. Springer. 2007, pp. 289–302.

[82] Ali Fakhrzadehgan et al. “SecDDR: Enabling low-cost secure memories by protecting the DDR interface”. In: *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2023, pp. 14–27.

[83] Newton Faller. “An adaptive system for data compression”. In: *Record of the 7-th Asilomar Conference on Circuits, Systems and Computers*. 1973, pp. 593–597.

[84] Panagiota Fatourou, Nikolaos D Kallimanis, and Eleftherios Kosmas. “The performance power of software combining in persistence”. In: *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2022, pp. 337–352.

[85] Erhu Feng et al. “Efficient Distributed Secure Memory with Migratable Merkle Tree”. In: *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2023, pp. 347–360.

[86] Alexander Freij, Huiyang Zhou, and Yan Solihin. “Bonsai merkle forests: Efficiently achieving crash consistency in secure persistent memory”. In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 2021, pp. 1227–1240.

[87] Alexander Freij, Huiyang Zhou, and Yan Solihin. “SecPB: Architectures for Secure Non-Volatile Memory with Battery-Backed Persist Buffers”. In: *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2023, pp. 677–690.

[88] Alexander Freij et al. “Persist level parallelism: Streamlining integrity tree updates for secure persistent memory”. In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020, pp. 14–27.

[89] Alexander Freij et al. “Streamlining integrity tree updates for secure persistent non-volatile memory”. In: *arXiv preprint arXiv:2003.04693* (2020).

[90] Robert Gallager. “Variations on a theme by Huffman”. In: *IEEE Transactions on Information Theory* 24.6 (1978), pp. 668–674.

[91] Juan A Garay, Vladimir Kolesnikov, and Rae Mclellan. “MAC precomputation with applications to secure memory”. In: *ACM Transactions on Privacy and Security (TOPS)* 19.2 (2016), pp. 1–21.

[92] Pablo García et al. “Beating the birthday paradox in dining cryptographer networks”. In: *Progress in Cryptology-LATINCRYPT 2014: Third International Conference on Cryptology and Information Security in Latin America Florianópolis, Brazil, September 17–19, 2014 Revised Selected Papers 3*. Springer. 2015, pp. 179–198.

[93] Blaise Gassend et al. “Caches and hash trees for efficient memory integrity verification”. In: *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* IEEE. 2003, pp. 295–306.

[94] Blaise Gassend et al. “Caches and merkle trees for efficient memory authentication”. In: (2002).

[95] Amir Gholami et al. “AI and memory wall”. In: *IEEE Micro* (2024).

[96] Tanguy Gilmont, J-D Legat, and J-J Quisquater. “Enhancing security in the memory management unit”. In: *Proceedings 25th EUROMICRO Conference. Informatics: Theory and Practice for the New Millennium.* Vol. 1. IEEE. 1999, pp. 449–456.

[97] Robert A Gingell et al. “Shared libraries in SunOS”. In: *AUUGN* 8.5 (1987), p. 112.

[98] Donghyun Gouk et al. “Memory pooling with cxl”. In: *IEEE Micro* 43.2 (2023), pp. 48–57.

[99] Shay Gueron. “A Memory Encryption Engine Suitable for General Purpose Processors”. In: *Cryptology ePrint Archive* (2016).

[100] David Gullasch, Endre Bangerter, and Stephan Krenn. “Cache games–bringing access-based cache attacks on AES to practice”. In: *2011 IEEE Symposium on Security and Privacy.* IEEE. 2011, pp. 490–505.

[101] Yunyan Guo and Guoliang Li. “A CXL-Powered Database System: Opportunities and Challenges”. In: *2024 IEEE 40th International Conference on Data Engineering (ICDE).* IEEE. 2024, pp. 5593–5604.

[102] J Alex Halderman et al. “Lest we remember: cold-boot attacks on encryption keys”. In: *Communications of the ACM* 52.5 (2009), pp. 91–98.

[103] W Eric Hall and Charanjit S Jutla. “Parallelizable authentication trees”. In: *Selected Areas in Cryptography: 12th International Workshop, SAC 2005, Kingston, ON, Canada, August 11-12, 2005, Revised Selected Papers 12.* Springer. 2006, pp. 95–109.

[104] Erik G Hallnor and Steven K Reinhardt. “A unified compressed memory hierarchy”. In: *11th International Symposium on High-Performance Computer Architecture.* IEEE. 2005, pp. 201–212.

[105] Greg Hamerly et al. “Simpoint 3.0: Faster and more flexible program phase analysis”. In: *Journal of Instruction Level Parallelism* 7.4 (2005), pp. 1–28.

[106] Richard W Hamming. “Error detecting and error correcting codes”. In: *The Bell system technical journal* 29.2 (1950), pp. 147–160.

[107] Xijing Han, James Tuck, and Amro Awad. “Dolos: Improving the performance of persistent applications in adr-supported secure memory”. In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture.* 2021, pp. 1241–1253.

[108] Xijing Han, James Tuck, and Amro Awad. “Thoth: Bridging the Gap Between Persistently Secure Memories and Memory Interfaces of Emerging NVMs”. In: *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA).* IEEE. 2023, pp. 94–107.

[109] Youngkwang Han and John Kim. “A novel covert channel attack using memory encryption engine cache”. In: *Proceedings of the 56th Annual Design Automation Conference 2019.* 2019, pp. 1–6.

[110] Reza Hashemian. “Memory efficient and high-speed search Huffman coding”. In: *IEEE Transactions on communications* 43.10 (2002), pp. 2576–2581.

[111] Donald Hepkin. “Active memory expansion: Overview and usage guide”. In: *IBM Corporation* (2010), pp. 1–25.

[112] Muhammad El-Hindi et al. “Benchmarking the second generation of intel sgx hardware”. In: *Proceedings of the 18th International Workshop on Data Management on New Hardware*. 2022, pp. 1–8.

[113] Takahiro Hirofuchi and Ryousei Takano. “A prompt report on the performance of Intel Optane DC persistent memory module”. In: *IEICE TRANSACTIONS on Information and Systems* 103.5 (2020), pp. 1168–1172.

[114] Michio Honda et al. “{PASTE}: A Network Programming Interface for {Non-Volatile} Main Memory”. In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 2018, pp. 17–33.

[115] Jeongmin Hong et al. “Bandwidth-Effective DRAM Cache for GPU s with Storage-Class Memory”. In: *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2024, pp. 139–155.

[116] Fangyong Hou et al. “Efficient encryption-authentication of shared bus-memory in SMP system”. In: *International Conference on Computer and Information Technology*. 2010, pp. 871–876.

[117] William C Hsieh and William E Weihl. “Scalable reader-writer locks for parallel systems”. In: (1991).

[118] David A Huffman. “A method for the construction of minimum-redundancy codes”. In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101.

[119] Ronen Aharon Hyatt. *Memory Mapping and CXL Translation to Exploit Unused Remote Memory in a Multi-Host System*. US Patent App. 18/611,472. Sept. 2024.

[120] Jisung Im et al. “Analysis of Row Hammer and Passing Gate Effect in DRAM Cells by BCAT Structural Design”. In: *2024 IEEE Silicon Nanoelectronics Workshop (SNW)*. IEEE. 2024, pp. 91–92.

[121] Akiko Inoue et al. “ELM: A low-latency and scalable memory encryption scheme”. In: *IEEE Transactions on Information Forensics and Security* 17 (2022), pp. 2628–2643.

[122] *Intel Awarded up to $3B by the Biden-Harris Administration for Secure Enclave*. `https://www.intel.com/content/www/us/en/newsroom/news/2024-intel-news.html`. Accessed: 10-10-2024.

[123] *Intel® Optane™ Persistent Memory 200 Series Brief*. `https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-persistent-memory-200-series-brief.html`. Accessed: 2023-04-27.

[124] Yutaka Ito and Yuan He. *Apparatus and methods for refreshing memory*. US Patent 10,490,252. Nov. 2019.

[125] Oleksandr Ivanov, Victor Ruzhentsev, and Roman Oliynykov. “Comparison of modern network attacks on TLS protocol”. In: *2018 International Scientific-Practical Conference Problems of Infocommunications. Science and Technology (PIC S&T)*. IEEE. 2018, pp. 565–570.

[126] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. “Failure-atomic persistent memory updates via JUSTDO logging”. In: *ACM SIGARCH Computer Architecture News* 44.2 (2016), pp. 427–442.

[127] Joseph Izraelevitz et al. “Basic performance measurements of the intel optane DC persistent memory module”. In: *arXiv preprint arXiv:1903.05714* (2019).

[128] Jae-Won Jang et al. “Self-correcting STTRAM under magnetic field attacks”. In: *Proceedings of the 52nd Annual Design Automation Conference*. 2015, pp. 1–6.

[129] Seth Jennings. “The zswap compressed swap cache”. In: *LWN. net* (2013).

[130] Simon Johnson et al. *Supporting Intel SGX on Multi-Socket Platforms*. Accessed: 2023-11-28. URL: %5Curl%7Bhttps://www.intel.com/content/dam/www/public/us/en/documents/white-papers/supporting-intel-sgx-on-mulit-socket-platforms.pdf%7D.

[131] Jonas Juffinger et al. “Presshammer: Rowhammer and Rowpress without Physical Address Information”. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2024, pp. 460–479.

[132] Myoungsoo Jung. “Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd)”. In: *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*. 2022, pp. 45–51.

[133] Eric Keller et al. “Nohype: virtualized cloud infrastructure without the virtualization”. In: *Proceedings of the 37th annual international symposium on Computer architecture*. 2010, pp. 350–361.

[134] Terence Kelly. “Persistent Memory Programming on Conventional Hardware: The persistent memory style of programming can dramatically simplify application software.” In: *Queue* 17.4 (2019), pp. 1–20.

[135] Ben Kenwright. “Fast efficient fixed-size memory pool: No loops and no overhead”. In: *arXiv preprint arXiv:2210.16471* (2022).

[136] Dongkyun Kim et al. “A 1.1-V 10-nm class 6.4-Gb/s/pin 16-Gb DDR5 SDRAM with a phase rotator-ILO DLL, high-speed SerDes, and DFE/FFE equalization scheme for Rx/Tx”. In: *IEEE Journal of Solid-State Circuits* 55.1 (2019), pp. 167–177.

[137] Jesung Kim et al. “Efficient Integrity-Tree Structure for Convolutional Neural Networks through Frequent Counter Overflow Prevention in Secure Memories”. In: *Sensors* 22.22 (2022), p. 8762.

[138] Yoongu Kim et al. “Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors”. In: *ACM SIGARCH Computer Architecture News* 42.3 (2014), pp. 361–372.

[139] Donald E Knuth. “Dynamic huffman coding”. In: *Journal of algorithms* 6.2 (1985), pp. 163–180.

[140] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. *HMAC: Keyed-hashing for message authentication*. Tech. rep. 1997.

[141] David Kroft. “Lockup-free instruction fetch/prefetch cache organization”. In: *25 years of the international symposia on Computer architecture (selected papers)*. 1998, pp. 195–201.

[142] Anil Kurmus et al. “From random block corruption to privilege escalation: A filesystem attack vector for rowhammer-like attacks”. In: *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. 2017.

[143] Wonok Kwon, Chanho Park, and Myeonghoon Oh. “Gen-z memory pool system architecture”. In: *2020 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE. 2020, pp. 1356–1360.

[144] Andres Lagar-Cavilla et al. “Software-defined far memory in warehouse-scale computers”. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019, pp. 317–330.

[145] Christoph Lameter. “NUMA (Non-Uniform Memory Access): An Overview: NUMA becomes more common because memory controllers get close to execution units on microprocessors.” In: *Queue* 11.7 (2013), pp. 40–51.

[146] Benjamin C Lee et al. “Phase change memory architecture and the quest for scalability”. In: *Communications of the ACM* 53.7 (2010), pp. 99–106.

[147] Jae Seong Lee, Piljoo Choi, and Dong Kyue Kim. “Lightweight and low-latency AES accelerator using shared SRAM”. In: *IEEE Access* 10 (2022), pp. 30457–30464.

[148] Jang-Soo Lee, Won-Kee Hong, and Shin-Dug Kim. “Design and evaluation of a selective compressed memory system”. In: *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No. 99CB37040)*. IEEE. 1999, pp. 184–191.

[149] Tamara Silbergleit Lehman. “Design Strategies for Efficient and Secure Memory”. PhD thesis. Duke University, 2019.

[150] Tamara Silbergleit Lehman, Andrew D Hilton, and Benjamin C Lee. “Maps: Understanding metadata access patterns in secure memory”. In: *2018 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE. 2018, pp. 33–43.

[151] Tamara Silbergleit Lehman, Andrew D Hilton, and Benjamin C Lee. “PoisonIvy: Safe speculation for secure memory”. In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2016, pp. 1–13.

[152] Mengya Lei et al. “SecNVM: An efficient and write-friendly metadata crash consistency scheme for secure NVM”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 19.1 (2021), pp. 1–26.

[153] Philip Levis, Kun Lin, and Amy Tai. “A case against cxl memory pooling”. In: *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*. 2023, pp. 18–24.

[154] Kyung-Suk Lhee and Steve J Chapin. “Buffer overflow and format string overflow vulnerabilities”. In: *Software: practice and experience* 33.5 (2003), pp. 423–460.

[155] Huaicheng Li et al. “Pond: Cxl-based memory pooling systems for cloud platforms”. In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 2023, pp. 574–587.

[156] Junjie Li et al. “SPEChpc 2021 benchmark suites for modern HPC systems”. In: *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*. 2022, pp. 15–16.

[157] Shang Li et al. “DRAMsim3: A cycle-accurate, thermal-capable DRAM simulator”. In: *IEEE Computer Architecture Letters* 19.2 (2020), pp. 106–109.

[158] Yuze Li and Shunyu Yao. “Understanding and Optimizing Serverless Workloads in CXL-Enabled Tiered Memory”. In: *arXiv preprint arXiv:2309.01736* (2023).

[159] David Lie, Chandramohan A Thekkath, and Mark Horowitz. "Implementing an untrusted operating system on trusted hardware". In: *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 2003, pp. 178–192.

[160] David Lie et al. "Architectural support for copy and tamper resistant software". In: *Acm Sigplan Notices* 35.11 (2000), pp. 168–177.

[161] David Lie et al. "Specifying and verifying hardware for tamper-resistant software". In: *2003 Symposium on Security and Privacy, 2003.* IEEE. 2003, pp. 166–177.

[162] Xiancheng Lin et al. "Research on the CXL Memory". In: *2024 IEEE 6th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*. Vol. 6. IEEE. 2024, pp. 1428–1432.

[163] *Linux Kernel Documentation*. `https://www.kernel.org/doc/html/v4.9/kernel-documentation.html`. Accessed: 2022-07-06.

[164] Helger Lipmaa, Phillip Rogaway, and David Wagner. "CTR-mode encryption". In: *First NIST Workshop on Modes of Operation*. Vol. 39. Citeseer. MD. 2000.

[165] Moses Liskov, Ronald L Rivest, and David Wagner. "Tweakable block ciphers". In: *Advances in Cryptology—CRYPTO 2002: 22nd Annual International Cryptology Conference Santa Barbara, California, USA, August 18–22, 2002 Proceedings 22*. Springer. 2002, pp. 31–46.

[166] Jie Liu et al. "Exploring and Evaluating Real-world CXL: Use Cases and System Adoption". In: *arXiv preprint arXiv:2405.14209* (2024).

[167] Jason Lowe-Power et al. "The gem5 simulator: Version 20.0+". In: *arXiv preprint arXiv:2007.03152* (2020).

[168] Haocong Luo et al. "Rowpress: Amplifying read disturbance in modern dram chips". In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 2023, pp. 1–18.

[169] Adrian Lutsch et al. "Benchmarking Analytical Query Processing in Intel SGXv2". In: *arXiv preprint arXiv:2403.11874* (2024).

[170] Lucas Matana Luza et al. "Emulating the effects of radiation-induced soft-errors for the reliability assessment of neural networks". In: *IEEE Transactions on Emerging Topics in Computing* 10.4 (2021), pp. 1867–1882.

[171] Lucas Matana Luza et al. "Neutron-induced effects on a self-refresh DRAM". In: *Microelectronics Reliability* 128 (2022), p. 114406.

[172] Umesh Maheshwari, Radek Vingralek, et al. "How to build a trusted database system on untrusted storage". In: *Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*. 2000.

[173] Evgeny Manzhosov and Simha Sethumadhavan. "Polymorphic Error Correction". In: *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2024, pp. 246–262.

[174] Janarbek Matai, Joo-Young Kim, and Ryan Kastner. "Energy efficient canonical huffman encoding". In: *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*. IEEE. 2014, pp. 202–209.

[175] Saurav Mathur, Nirmit Jallawar, and Swamit Tannu. "Flipping Bits in DRAM using Laser Induced Localized Heating". In: ().

[176] Ueli M Maurer and Stefan Wolf. "The diffie–hellman protocol". In: *Designs, Codes and Cryptography* 19.2 (2000), pp. 147–171.

[177] David McGrew and John Viega. "The Galois/counter mode of operation (GCM)". In: *submission to NIST Modes of Operation Process* 20 (2004), pp. 0278–0070.

[178] Ralph C Merkle. "A digital signature based on a conventional encryption function". In: *Conference on the theory and application of cryptographic techniques*. Springer. 1987, pp. 369–378.

[179] Ralph C Merkle. "One way hash functions and DES". In: *Conference on the Theory and Application of Cryptology*. Springer. 1989, pp. 428–446.

[180] Ralph Charles Merkle. *Secrecy, authentication, and public key systems.* Stanford university, 1979.

[181] RC MERKLE. "Protocols for Public Key Cryptosystems". In: *IEEE Symposium on Security and Privacy, 1980*. 1980, pp. 122–134.

[182] Matthew Millar, Marcin Łukowiak, and Stanislaw Radziszowski. "Memory Protection with Dynamic Authentication Trees". In: *2022 29th International Conference on Mixed Design of Integrated Circuits and System (MIXDES)*. IEEE. 2022, pp. 202–207.

[183] Matthew T Millar. *Flexible Memory Protection with Dynamic Authentication Trees*. Rochester Institute of Technology, 2021.

[184] Dong-Sun Min Dong-Sun Min et al. "Wordline coupling noise reduction techniques for scaled DRAMs". In: *Digest of Technical Papers., 1990 Symposium on VLSI Circuits*. IEEE. 1990, pp. 81–82.

[185] Sparsh Mittal and Jeffrey S Vetter. "A survey of software techniques for using non-volatile memories for storage and main memory systems". In: *IEEE Transactions on Parallel and Distributed Systems* 27.5 (2015), pp. 1537–1550.

[186] Yilin Mo and Bruno Sinopoli. "Secure control against replay attacks". In: *2009 47th annual Allerton conference on communication, control, and computing (Allerton)*. IEEE. 2009, pp. 911–918.

[187] Alistair Moffat. "Huffman coding". In: *ACM Computing Surveys (CSUR)* 52.4 (2019), pp. 1–35.

[188] Janani Mukundan et al. "Understanding and mitigating refresh overheads in high-density DDR4 DRAM systems". In: *ACM SIGARCH Computer Architecture News* 41.3 (2013), pp. 48–59.

[189] Richard C Murphy et al. "Introducing the graph 500". In: *Cray Users Group (CUG)* 19.45-74 (2010), p. 22.

[190] Onur Mutlu. "The RowHammer problem and other issues we may face as memory becomes denser". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE. 2017, pp. 1116–1121.

[191] Onur Mutlu and Jeremie S Kim. "Rowhammer: A retrospective". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.8 (2019), pp. 1555–1571.

[192] Seonjin Na et al. "Common counters: Compressed encryption counters for secure GPU memory". In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2021, pp. 1–13.

[193] Seonjin Na et al. “Supporting Secure Multi-GPU Computing with Dynamic and Batched Metadata Management”. In: *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2024, pp. 204–217.

[194] Prashant J Nair, Bahar Asgari, and Moinuddin K Qureshi. “SuDoku: Tolerating high-rate of transient failures for enabling scalable STTRAM”. In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2019, pp. 388–400.

[195] Doron Nakar and Shlomo Weiss. “Selective main memory compression by identifying program phase changes”. In: *Proceedings of the 3rd workshop on Memory performance issues: in conjunction with the 31st international symposium on computer architecture*. 2004, pp. 96–101.

[196] Sanketh Nalli et al. “An analysis of persistent memory use with WHISPER”. In: *ACM SIGPLAN Notices* 52.4 (2017), pp. 135–148.

[197] Mihir Nanavati et al. “Non-volatile storage”. In: *Communications of the ACM* 59.1 (2015), pp. 56–63.

[198] Mehrdad Nourani and Mohammad H Tehranipour. “RL-Huffman encoding for test compression and power reduction in scan applications”. In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 10.1 (2005), pp. 91–115.

[199] Stanko Novakovic et al. “Mitigating load imbalance in distributed data serving with rack-scale memory pooling”. In: *ACM Transactions on Computer Systems (TOCS)* 36.2 (2019), pp. 1–37.

[200] *NVLink High-Speed GPU Interconnect*. `https://www.nvidia.com/en-us/design-visualization/nvlink-bridges/`. Accessed: 19-07-2024.

[201] Anas Al-Okaily et al. “Toward a better compression for DNA sequences using Huffman encoding”. In: *Journal of Computational Biology* 24.4 (2017), pp. 280–288.

[202] Geraldo F Oliveira et al. “Extending memory capacity in consumer devices with emerging non-volatile memory: An experimental study”. In: *arXiv preprint arXiv:2111.02325* (2021).

[203] Elliott I Organick. *The Multics system: an examination of its structure*. MIT press, 1972.

[204] Steven Pelley, Peter M Chen, and Thomas F Wenisch. “Memory persistency”. In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE. 2014, pp. 265–276.

[205] Erez Perelman et al. “Using simpoint for accurate and efficient simulation”. In: *ACM SIGMETRICS Performance Evaluation Review* 31.1 (2003), pp. 318–319.

[206] Nicole Perlroth. “Software Meant to Fight Crime Is Used to Spy on Dissidents”. In: *New York Times* 30 (2012).

[207] Steven Pigeon and Yoshua Bengio. “A memory-efficient Huffman adaptive coding algorithm for very large sets of symbols”. In: *Université de Montréal, Rapport technique* 1081 (1997).

[208] Benny Pinkas and Tzachy Reinman. “Oblivious RAM revisited”. In: *Advances in Cryptology–CRYPTO 2010: 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings 30*. Springer. 2010, pp. 502–519.

[209] Sandro Pinto and Nuno Santos. “Demystifying arm trustzone: A comprehensive survey”. In: *ACM computing surveys (CSUR)* 51.6 (2019), pp. 1–36.

[210] pmem.io. *Persistent Memory Development Kit*. `http://pmem.io/pmdk`. 2017.

[211] NF Pub. “Specification for the advanced encryption standard (AES)”. In: *Tech. Rep. FIPS PUB 197, Federal Information Processing Standards* (2001).

[212] Moinuddin Qureshi. “Rethinking ECC in the era of row-hammer”. In: *DRAMSec*. 2021.

[213] Adnan Siraj Rakin et al. “Deepsteal: Advanced model extractions leveraging efficient weight stealing in memories”. In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2022, pp. 1157–1174.

[214] Joydeep Rakshit and Kartik Mohanram. “Assure: Authentication scheme for secure energy efficient non-volatile memories”. In: *Proceedings of the 54th Annual Design Automation Conference 2017*. 2017, pp. 1–6.

[215] Luiz E Ramos, Eugene Gorbatov, and Ricardo Bianchini. “Page placement in hybrid memory systems”. In: *Proceedings of the international conference on Supercomputing*. 2011, pp. 85–95.

[216] Mark Randolph and William Diehl. “Power side-channel attack analysis: A review of 20 years of study for the layman”. In: *Cryptography* 4.2 (2020), p. 15.

[217] Michael Redeker, Bruce F Cockburn, and Duncan G Elliott. “An investigation into crosstalk noise in DRAM structures”. In: *Proceedings of the 2002 IEEE International Workshop on Memory Technology, Design and Testing (MTDT2002)*. IEEE. 2002, pp. 123–129.

[218] Edwin M Robertson. “Memory leaks: information shared across memory systems”. In: *Trends in cognitive sciences* 26.7 (2022), pp. 544–554.

[219] Giampaolo Rodola. *Psutil documentation*. 2020.

[220] Phillip Rogaway, Mihir Bellare, and John Black. “OCB: A block-cipher mode of operation for efficient authenticated encryption”. In: *ACM Transactions on Information and System Security (TISSEC)* 6.3 (2003), pp. 365–403.

[221] Brian Rogers et al. “Single-level integrity and confidentiality protection for distributed shared memory multiprocessors”. In: *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. IEEE. 2008, pp. 161–172.

[222] Brian Rogers et al. “Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly”. In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE. 2007, pp. 183–196.

[223] Andy Rudoff. “Persistent memory programming”. In: *Login: The Usenix Magazine* 42.2 (2017), pp. 34–40.

[224] Gururaj Saileshwar et al. “Morphable counters: Enabling compact integrity trees for low-overhead secure memories”. In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2018, pp. 416–427.

[225] Gururaj Saileshwar et al. “Synergy: Rethinking secure-memory design for error-correcting memories”. In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2018, pp. 454–465.

[226] Fernand Lone Sang, Vincent Nicomette, and Yves Deswarte. “I/O attacks in Intel PC-based architectures and countermeasures”. In: *2011 First SysSec Workshop*. IEEE. 2011, pp. 19–26.

[227] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. "ZeroTrace: Oblivious memory primitives from Intel SGX". In: *Cryptology ePrint Archive* (2017).

[228] Russ Sevinsky. "Funderbolt: Adventures in thunderbolt DMA attacks". In: *Black Hat USA* (2013).

[229] Rakin Muhammad Shadab, Yu Zou, and Mingjie Lin. "CTR+: A High-Performance Metadata Access Scheme for Secure Embedded Memory in Heterogeneous Computing Systems". In: *2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE. 2024, pp. 304–308.

[230] Rakin Muhammad Shadab et al. "Hmt: A hardware-centric hybrid bonsai merkle tree algorithm for high-performance authentication". In: *ACM Transactions on Embedded Computing Systems* 22.4 (2023), pp. 1–28.

[231] Rakin Muhammad Shadab et al. "OMT: A Run-time Adaptive Architectural Framework for Bonsai Merkle Tree-Based Secure Authentication with Embedded Heterogeneous Memory". In: *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE. 2023, pp. 191–202.

[232] Ali Shafiee et al. "Secure DIMM: Moving ORAM primitives closer to memory". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2018, pp. 428–440.

[233] Rifat Shahriyar et al. "Taking off the gloves with reference counting Immix". In: *ACM SIGPLAN Notices* 48.10 (2013), pp. 93–110.

[234] Debendra Das Sharma. "PCI Express® 6.0 Specification at 64.0 GT/s with PAM-4 signaling: a low latency, high bandwidth, high reliability and cost-effective interconnect". In: *2020 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE. 2020, pp. 1–8.

[235] Weidong Shi and Hsien-Hsin S Lee. "Accelerating memory decryption and authentication with frequent value prediction". In: *Proceedings of the 4th international conference on Computing frontiers*. 2007, pp. 35–46.

[236] Weidong Shi et al. "Architectural support for high speed protection of memory integrity and confidentiality in multiprocessor systems". In: *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.* IEEE. 2004, pp. 123–134.

[237] Weidong Shi et al. "High efficiency counter mode security architecture via prediction and precomputation". In: *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE. 2005, pp. 14–24.

[238] Ofir Shwartz and Yitzhak Birk. "Distributed memory integrity trees". In: *IEEE Computer Architecture Letters* 17.2 (2018), pp. 159–162.

[239] Sergei Skorobogatov. "Data remanence in flash memory devices". In: *Cryptographic Hardware and Embedded Systems–CHES 2005: 7th International Workshop, Edinburgh, UK, August 29–September 1, 2005. Proceedings 7*. Springer. 2005, pp. 339–353.

[240] Sergei Skorobogatov. "Physical attacks and tamper resistance". In: *Introduction to Hardware Security and Trust*. Springer, 2011, pp. 143–173.

[241] Rajeev Sobti and Ganesan Geetha. "Cryptographic hash functions: a review". In: *International Journal of Computer Science Issues (IJCSI)* 9.2 (2012), p. 461.

[242] Sure Srikanth and Sukadev Meher. "Compression efficiency for combining different embedded image compression techniques with Huffman encoding". In: *2013 International Conference on Communication and Signal Processing*. IEEE. 2013, pp. 816–820.

[243] Julian Stecklina. "Remote debugging via firewire". In: *Master's thesis, TU Dresden* (2009).

[244] Patrick Stewin and Iurii Bystrov. "Understanding DMA malware". In: *Detection of Intrusions and Malware, and Vulnerability Assessment: 9th International Conference, DIMVA 2012, Heraklion, Crete, Greece, July 26-27, 2012, Revised Selected Papers 9*. Springer. 2013, pp. 21–41.

[245] Joshua Suetterlein, Joseph Manzano, and Andres Marquez. "Synchronization for CXL Based Memory". In: *Proceedings of the International Symposium on Memory Systems*. 2024, pp. 178–185.

[246] G Edward Suh, Charles W O'Donnell, and Srinivas Devadas. "AEGIS: A single-chip secure processor". In: *Information Security Technical Report* 10.2 (2005), pp. 63–73.

[247] G Edward Suh et al. "AEGIS: Architecture for tamper-evident and tamper-resistant processing". In: *ACM International Conference on Supercomputing 25th Anniversary Volume*. 2003, pp. 357–368.

[248] G Edward Suh et al. "Efficient memory integrity verification and encryption for secure processors". In: *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* IEEE. 2003, pp. 339–350.

[249] Michael B Sullivan et al. "Implicit memory tagging: No-overhead memory safety using alias-free tagged ecc". In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 2023, pp. 1–13.

[250] Yan Sun et al. "Demystifying cxl memory with genuine cxl-ready systems and devices". In: *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 2023, pp. 105–121.

[251] Toyotaro Suzumura et al. "Performance characteristics of Graph500 on large-scale distributed environment". In: *2011 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2011, pp. 149–158.

[252] Shivam Swami and Kartik Mohanram. "ACME: Advanced counter mode encryption for secure non-volatile memories". In: *Proceedings of the 55th Annual Design Automation Conference*. 2018, pp. 1–6.

[253] Shivam Swami and Kartik Mohanram. "ASSET: Architectures for Smart Security of Non-Volatile Memories". In: *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2019, pp. 348–353.

[254] Shivam Swami and Kartik Mohanram. "COVERT: Counter OVErflow ReducTion for efficient encryption of non-volatlle memories". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE. 2017, pp. 906–909.

[255] Jakub Szefer and Sebastian Biedermann. "Towards fast hardware memory integrity checking with skewed merkle trees". In: *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*. 2014, pp. 1–8.

[256] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. "VAULT: Reducing paging overheads in SGX with efficient integrity verification structures". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 2018, pp. 665–678.

[257] Meysam Taassori et al. "Compact leakage-free support for integrity and reliability". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2020, pp. 735–748.

[258] Chengsong Tan, Alastair F Donaldson, and John Wickerson. "Formalising CXL Cache Coherence". In: *arXiv preprint arXiv:2410.15908* (2024).

[259] Wenda Tang et al. "Yggdrasil: Reducing Network I/O Tax with (CXL-Based) Distributed Shared Memory". In: *Proceedings of the 53rd International Conference on Parallel Processing*. 2024, pp. 597–606.

[260] Samuel Thomas et al. "A Midsummer Night's Tree: Efficient and High Performance Secure SCM". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 2024, pp. 22–37.

[261] Samuel Thomas et al. "Baobab Merkle Tree for Efficient Secure Memory". In: *IEEE Computer Architecture Letters* (2024).

[262] Samuel Thomas et al. "Rethinking Metadata Caches in Secure NVMs". In: ().

[263] Samuel Thomas et al. "Using skip graphs for increased NUMA locality". In: *Journal of Parallel and Distributed Computing* 167 (2022), pp. 31–49.

[264] R Brett Tremaine et al. "IBM memory expansion technology (MXT)". In: *IBM Journal of Research and Development* 45.2 (2001), pp. 271–285.

[265] Florian Tschorsch and Björn Scheuermann. "Bitcoin and beyond: A technical survey on decentralized digital currencies". In: *IEEE Communications Surveys & Tutorials* 18.3 (2016), pp. 2084–2123.

[266] James Tuck, Luis Ceze, and Josep Torrellas. "Scalable cache miss handling for high memory-level parallelism". In: *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE. 2006, pp. 409–422.

[267] James M Turner. "The keyed-hash message authentication code (hmac)". In: *Federal Information Processing Standards Publication* 198.1 (2008), pp. 1–13.

[268] *UALink will be the NVLink Standard Backed by AMD Intel Broadcom Cisco and More*. `https://www.servethehome.com/ualink-will-be-the-nvlink-standard-backed-by-amd-intel-broadcom-cisco-and-more/`. Accessed: 19-07-2024.

[269] Martin Unterguggenberger et al. "Cryptographically Enforced Memory Safety". In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2023, pp. 889–903.

[270] Thomas Unterluggauer and Stefan Mangard. "Exploiting the physical disparity: Side-channel attacks on memory encryption". In: *Constructive Side-Channel Analysis and Secure Design: 7th International Workshop, COSADE 2016, Graz, Austria, April 14-15, 2016, Revised Selected Papers 7*. Springer. 2016, pp. 3–18.

[271] Thomas Unterluggauer, Mario Werner, and Stefan Mangard. "MEAS: Memory encryption and authentication secure against side-channel attacks". In: *Journal of cryptographic engineering* 9 (2019), pp. 137–158.

[272] Kenzo Van Craeynest and Lieven Eeckhout. "The multi-program performance model: debunking current practice in multi-core simulation". In: *2011 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2011, pp. 26–37.

[273] Saru Vig, Guiyuan Jiang, and Siew-Kei Lam. "Dynamic skewed tree for fast memory integrity verification". In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2018, pp. 642–647.

[274] Saru Vig, Rohan Juneja, and Siew-Kei Lam. "DISSECT: dynamic skew-and-split tree for memory authentication". In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2020, pp. 1538–1543.

[275] Saru Vig, Siew-Kei Lam, and Rohan Juneja. "Cache-aware dynamic skewed tree for fast memory authentication". In: *Proceedings of the 26th Asia and South Pacific Design Automation Conference*. 2021, pp. 402–407.

[276] Saru Vig et al. "Customizing skewed trees for fast memory integrity verification in embedded systems". In: *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2017, pp. 213–218.

[277] Saru Vig et al. "Framework for fast memory authentication using dynamically skewed integrity tree". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.10 (2019), pp. 2331–2343.

[278] Saru Vig et al. "Rapid detection of Rowhammer attacks using dynamic skewed hash tree". In: *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*. 2018, pp. 1–8.

[279] Jeffrey Scott Vitter. "Design and analysis of dynamic Huffman codes". In: *Journal of the ACM (JACM)* 34.4 (1987), pp. 825–845.

[280] Haris Volos, Andres Jaan Tack, and Michael M Swift. "Mnemosyne: Lightweight persistent memory". In: *ACM SIGARCH Computer Architecture News* 39.1 (2011), pp. 91–104.

[281] Eric Wagner et al. "BP-MAC: fast authentication for short messages". In: *Proceedings of the 15th ACM conference on security and privacy in wireless and mobile networks*. 2022, pp. 201–206.

[282] Jacob Wahlgren, Maya Gokhale, and Ivy B Peng. "Evaluating emerging CXL-enabled memory pooling for HPC systems". In: *2022 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE. 2022, pp. 11–20.

[283] Carl A Waldspurger. "Memory resource management in VMware ESX server". In: *ACM SIGOPS Operating Systems Review* 36.SI (2002), pp. 181–194.

[284] Bolin Wang. "Bo-tree: a dynamic merkle tree for enabling scalable memories". PhD thesis. University of British Columbia, 2022.

[285] Liang Wang et al. "Side-channel attacks on shared search indexes". In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, pp. 673–692.

[286] Xin Wang, Jagadish B Kotra, and Xun Jian. "Eager memory cryptography in caches". In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2022, pp. 693–709.

[287] Xin Wang et al. "Counter-light Memory Encryption". In: *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2024, pp. 724–738.

[288] Xin Wang et al. "Self-Reinforcing Memoization for Cryptography Calculations in Secure Memory Systems". In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2022, pp. 678–692.

[289] Yanjing Wang et al. "A Comprehensive Simulation Framework for CXL Disaggregated Memory". In: *arXiv preprint arXiv:2411.02282* (2024).

[290] Gregg D Wolff. *Word line cache mode*. US Patent 10,366,733. July 2019.

[291] H-S Philip Wong et al. "Phase change memory". In: *Proceedings of the IEEE* 98.12 (2010), pp. 2201–2227.

[292] Yubin Xia, Yutao Liu, and Haibo Chen. "Architecture support for guest-transparent VM protection from untrusted hypervisor and physical attacks". In: *Proc. International Symposium on High Performance Computer Architecture (HPCA)*. 2013.

[293] Lingfeng Xiang et al. "Characterizing the performance of intel optane persistent memory: A close look at its on-dimm buffering". In: *Proceedings of the Seventeenth European Conference on Computer Systems*. 2022, pp. 488–505.

[294] Yang Xiao et al. "A survey of distributed consensus protocols for blockchain networks". In: *IEEE Communications Surveys & Tutorials* 22.2 (2020), pp. 1432–1465.

[295] Yuan Xiao et al. "One bit flips, one cloud flops:{Cross-VM} row hammer attacks and privilege escalation". In: *25th USENIX security symposium (USENIX Security 16)*. 2016, pp. 19–35.

[296] Yi Xu et al. "Position: CXL Shared Memory Programming: Barely Distributed and Almost Persistent". In: *arXiv preprint arXiv:2405.19626* (2024).

[297] Chun Jason Xue et al. "Emerging non-volatile memories: Opportunities and challenges". In: *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. 2011, pp. 325–334.

[298] Chenyu Yan et al. "Improving cost, performance, and security of memory encryption and authentication". In: *ACM SIGARCH Computer Architecture News* 34.2 (2006), pp. 179–190.

[299] Jun Yang, Youtao Zhang, and Lan Gao. "Fast secure processor for inhibiting software piracy and tampering". In: *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* IEEE. 2003, pp. 351–360.

[300] Yuval Yarom and Katrina Falkner. "{FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack". In: *23rd USENIX security symposium (USENIX security 14)*. 2014, pp. 719–732.

[301] Mao Ye, Clayton Hughes, and Amro Awad. *Osiris: A Low-Cost Mechanism to Enable Restoration of Secure Non-Volatile Memories.* Tech. rep. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2018.

[302] Zhisheng Ye et al. “Deep learning workload scheduling in gpu datacenters: A survey”. In: *ACM Computing Surveys* 56.6 (2024), pp. 1–38.

[303] Keun Soo Yim. “The rowhammer attack injection methodology”. In: *2016 IEEE 35th symposium on reliable distributed systems (SRDS)*. IEEE. 2016, pp. 1–10.

[304] Salessawi Ferede Yitbarek et al. “Exploring specialized near-memory processing for data intensive operations”. In: *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2016, pp. 1449–1452.

[305] Jonathan Yu and Koorosh Aflatooni. “Leakage current in dram memory cell”. In: *2006 16th Biennial University/Government/Industry Microelectronics Symposium*. IEEE. 2006, pp. 191–194.

[306] Yang Yu, Sha Tao, and Elena Dubrova. “Comparison of CRC and KECCAK Based Message Authentication for Resource-Constrained Devices”. In: *2018 16th IEEE International New Circuits and Systems Conference (NEWCAS)*. IEEE. 2018, pp. 217–220.

[307] Shougang Yuan, Amro Awad, and Huiyang Zhou. “Delta Counter: Bandwidth-Efficient Encryption Counter Representation for Secure GPU Memory”. In: *IEEE Transactions on Dependable and Secure Computing* (2024).

[308] Shougang Yuan, Yan Solihin, and Huiyang Zhou. “Pssm: Achieving secure memory for gpus with partitioned and sectored security metadata”. In: *Proceedings of the ACM International Conference on Supercomputing*. 2021, pp. 139–151.

[309] Shougang Yuan et al. “Adaptive security support for heterogeneous memory on gpus”. In: *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2022, pp. 213–228.

[310] Shougang Yuan et al. “Analyzing secure memory architecture for gpus”. In: *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2021, pp. 59–69.

[311] Mingxing Zhang et al. “Partial failure resilient memory management system for (cxl-based) distributed shared memory”. In: *Proceedings of the 29th Symposium on Operating Systems Principles*. 2023, pp. 658–674.

[312] ChongChong Zhao et al. “On the performance of intel sgx”. In: *2016 13Th web information systems and applications conference (WISA)*. IEEE. 2016, pp. 184–187.

[313] Jishen Zhao et al. “Kiln: Closing the performance gap between systems with and without persistence support”. In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 2013, pp. 421–432.

[314] Wenting Zheng et al. “Opaque: An oblivious and encrypted distributed analytics platform”. In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 2017, pp. 283–298.

[315] Jacob Ziv and Abraham Lempel. “A universal algorithm for sequential data compression”. In: *IEEE Transactions on information theory* 23.3 (1977), pp. 337–343.

[316] Yu Zou and Mingjie Lin. “Fast: A frequency-aware skewed merkle tree for fpga-secured embedded systems”. In: *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2019, pp. 326–331.

[317] Kazi Abu Zubair and Amro Awad. “Anubis: ultra-low overhead and recovery time for secure non-volatile memories”. In: *Proceedings of the 46th International Symposium on Computer Architecture*. 2019, pp. 157–168.

[318] Pengfei Zuo, Yu Hua, and Yuan Xie. “Supermem: Enabling application-transparent secure persistent memory with low overheads”. In: *IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 479–492.