Abstract of "Proposal: A Study of Performance and Trust in Secure Memory" by Samuel Thomas, Ph.D., Brown University, Fall 2023.

We are currently in the midst of a wave a renewed interest in secure memory, as emerging technologies introduce new questions of practical integration with commodity devices. However, fundamental questions about *performance* with secure memory remain unanswered, particularly as the baseline secure memory protocol adapts to these modern devices. In this thesis, I examine case studies of secure memory applied to emerging technologies. I will describe a series of implementations and optimizations to the secure memory protocol for each of these cases. In particular, I will examine: how non-volatile main memories require a crash consistency scheme for secure memory metadata and how global address spaces across multiple devices with CXL impacts the coherence of secure memory metadata. The protocols and micro-architectures described in this thesis reduce both runtime overhead *and* on-chip area overhead, which has previously been neglected by the state-of-the-art.

In answering questions of performance, this thesis will also re-introduce *trust* as a first-class design issue in secure memory. It is no secret that security guarantees often come at the expense of performance. However, in this thesis I introduce the question of whether key optimizations to the secure memory protocol may lead to potential unforeseen consequences. The Trusted Computing Base (TCB) was thoroughly discussed in the early wave of secure memory, but has fallen to the wayside in the implementation and device driven era of secure memory. As a result, many modern secure memory protocols have bloated TCBs, which may lead to future vulnerabilities. In this thesis I will also consider secure memory protocols that are optimized for performance, but treat trust as a variable design feature. I will describe how trust influences protocols both in distributed and local secure memory, and will invoke a discussion of the implications of both.

For the purposes of this proposal, I will also describe my goals for the rest of my Ph.D. in terms of 3 month, 9 month, and 18 month aspirations.

Proposal: A Study of Performance and Trust in Secure Memory

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
Fall 2023

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 _____ _____

Dr. R. Iris Bahar, Director

Recommended to the Graduate Council

Date _____ _____

Dr. Maurice Herlihy, Reader

Date _____ _____

Dr. Malte Schwarzkopf, Reader

Date _____ _____

Dr. Tamara Lehman, Reader

Approved by the Graduate Council

Date _____ _____

Dean of the Graduate School

Vita

It all started in a little log cabin ...

Preface

This thesis tells you all you need to know about...

Acknowledgements

I would like to thank the academy...

Contents

List of Tables

List of Figures

Part I

Introduction and Background

Chapter 1

Introduction

Secure memory describes a protocol proposed back in the 1990s [10, 35, 60]. Securing memory devices is particularly interesting to the architecture security community because of several key physical properties about memory. For one, memory is attached to the processor via bus rather than being wired directly or fabricated onto the motherboard. As such, the device itself is easily detachable, replaceable, and is subject to tampering with minimal effort from an attacker. This means that attackers, who merely have physical access to a device, can easily read and corrupt memory contents.

Furthermore, main memory is built on top of the most vulnerable physical device is the memory hierarchy. Cells are typically composed of large DRAM arrays (on the order of several gigabytes). A DRAM cell can be composed of a single capacitor and transfer device [55], but due to its far simpler design than alternative technologies (i.e., SRAM caches, flash-based SSDs etc...) variations in environment can be maliciously leveraged to modify the state of stored data. For example, Rowhammer [53] attacks flip bits in main memory without actually accessing those addresses, which means that software-level defenses like access permissions are insufficient defenses. As such, much work went into hardware driven solutions to secure memory devices in the early 2000s.

Two key design features drove the conversation of securing values in memory around this time period – data encryption [63, 28, 65, 99] and integrity protection [34, 59, 61, 101, 99]. These works demonstrated a clear direction that the architecture needed to provide support for defenses against this range of vulnerabilities.

Implementing both confidentiality and/or integrity over data in memory requires *security metadata*. To provide the integrity for some data, an auxiliary piece of information must be stored alongside the data to ensure that it hasn't been tampered with, like a hash. The hardware then needs to fetch both the data and the associated metadata. Similarly, to encrypt/decrypt data in memory, a temporally and spatially unique counter value needs to be stored in memory alongside the data. This counter needs to be fetched and is used as input to an AES engine to perform the cipher. Each of these protocols are described in more detail in Chapter 2. These additional fetches occupy bandwidth over the insecure alternatives, and the cipher and/or check requires some additional computation cycles post fetch.

Into the 2010s, these theoretical solutions manifested themselves in commodity devices from industry. Intel developed Software Guard Extensions (SGX) in 2015 [8], which entailed many user-level security protections guaranteed by the hardware, including secure memory [26]. Around the same time, ARM released

TrustZone [24], which provided hardware supported attestation and authentication that values are tamper-free [72]. Both of these products demonstrated the desire in the market for these architecture defenses, and both also demonstrate the performance challenges of having a usable product on the market. Intel produced SGX version 2 as a response to market feedback from the original SGX [67]. The new product made many changes, including cuts to the guarantees of their integrity protection mechanism in favor of performance. Meanwhile, the integrity guarantees in TrustZone were never as comprehensive as those made in SGX for the sake of performance.

The implicit acknowledgements of performance limitations from industry along with the development of emerging technologies that challenge traditional secure memory semantics provided the framework for a new wave of research in the secure memory space. Performance became a first-class design problem in developing protocols for secure non-volatile memory [103, 106, 5], distributed secure memory [89, 6], ADR-backed secure memory [37, 45, 38, 39], etc.

In this thesis, I will describe my contributions in continuing the progress towards improving performance in secure memory. I will also re-introduce important questions about trust in the usage of secure memory, and in particular consider the impact of how trust may change across different use cases of secure memory systems.

The organization of the rest of this document is as follows:

1. In the rest of Part 1, I will describe the background of secure memory in more detail. I will describe each component of the protocol, describe the related work that was foundational to that component, and perform a sensitivity analysis across each of the hyper-parameters in the secure memory protocol.

2. In Part 2, I will work to describe several of the proposed models in which I have tried to challenge the existing secure memory model, what has proven to be effective, what has not, and provide an insight into why. Within this, I will describe three key ideas: (1) taking advantage of subtrees within the integrity tree; (2) memoizing encryption counter values; and (3) Huffmanizing the integrity tree.

3. In Part 3, I will describe the potential impacts of use cases of secure memory on trust and vice versa. In particular, I will describe how trust can guide optimizations both in local and distributed settings under a varieties of trust models ranging from high paranoia to high trust.

Chapter 2

Background

In this chapter, I will describe the secure memory protocol in depth. In particular, I will describe how the protocol enforces privacy through encryption, integrity through authentication, and the different proposed variations across this trade-off space. Then, I will demonstrate an in-depth sensitivity analysis across the key features of the secure memory protocol to demonstrate the true overheads of different components of the protocol by toggling the hyper-parameters of the protocol. This sensitivity study will provide the reader with an insight into the true overheads of each feature in the protocol.

2.1 Threat Model

Recent literature in the secure memory community generally assume a consistent threat model within a single memory device [5, 6, 3, 11, 12, 20, 29, 33, 32, 40, 37, 38, 39, 57, 56, 80, 81]. They assume a capable attacker with physical access to a device, and who can run legitimate processes with user permissions on the device. Within this, the hardware should defend against an attacker who can read values in memory and can precisely modify values. That is, any integrity guarantees made by the hardware should be able to defend against all splicing and spoofing attacks on values in memory, as well as against replay attacks.

Memory is of particular interest due to several important physical characteristics and properties. Unlike caches, registers, on-chip interconnects, processor buffers, and other on-chip components, main memory is attached to the memory *externally* via the memory bus. This means that an attacker with physical access to a device can detach the device from a machine, attach it to some other device and directly modify values in memory. Furthermore, software defenses are insufficient to protect main memory. Unlike processor chips, which are small and extraordinarily dense, main memory is a large array, and is highly exposed in the layout of the device. Comparable tampering of precise values in on-chip components would require a much more capable attacker.

Beyond the layout of main memory, securing memory is an interesting challenge due to the physical characteristics of main memory. In particular, most traditional main memories are fabricated out of DRAM CMOS technology – which is fabricated out of transistors that have an unstable circuit and requires frequent power refreshes in order to retain its state. Depending on whether or not the transistor is in high-power or low-power state determines the state of a bit. However, the mechanism of this technology has an exploitable unintended consequence. Due to the close proximity or transistors in DRAM arrays, frequently storing the

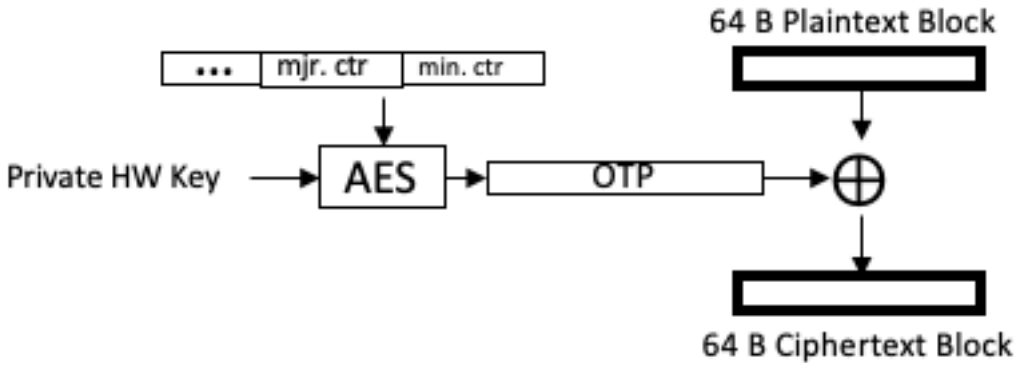


Figure 2.1: TODO: remake this figure (but something like this).

value to a single transistor creates leakage charge across the array. In high quantities, this leakage enforces a similar effect to setting the value of a transistor elsewhere in the array that wasn't actually accessed. This attack, called Rowhammer [70], is particular to DRAM and as such is a memory specific problem.

Seeing as attackers can legitimately run processes on the victim machine, it is worth mentioning that there are certain software vulnerabilities that the attacker can exploit without necessarily taking advantage of the underlying architecture. For example, an attacker may exploit buffer overflow vulnerabilities, perform a phishing attack, etc. These attacks and vulnerabilities are important to defend against, but are out-of-scope for this line of research. Instead, this work is concerned with ensuring that the underlying architecture does not suffer from exploitable vulnerabilities upon which sensitive software might run. If we can develop hardware that provides guarantees of confidentiality and integrity at the memory level, then the capability required of an attacker to exploit the underlying architecture increases significantly.

2.2 Privacy

In order for a system to achieve confidentiality of values in memory, these values *cannot* reside in memory in plain-text. If they do, then an attacker with physical access to the device can simply detach the memory device, attach it to another device, and read the values thereby violating the confidentiality of values in memory. As such, data values need to be *encrypted* by the memory controller before being sent off-chip to the memory device.

There are multiple means by which data can be encrypted in hardware. The one that is typically used in the secure memory literature is to use counter-mode encryption [66, 91, 65, 100] (CME). It describes an encryption/decryption unit located between the LLC and the main memory unit. On eviction or writeback, values are encrypted prior to storage into off-chip devices and are subsequently decrypted on load back into the on-chip cache hierarchy [100, 41]. CME has several advantages that have led to its popularity – it uses a unique counter value per each 64B word in memory which is incremented on every memory write. As a result, the encryption counter associated with each word of memory is both temporally and spatially unique. That is, the encryption seed varies by address and will change over time.

In general, CME works by fetching a counter value from memory that is unique to each data. Once the counter is fetched, the counter (along with other metadata fields unique to the data like page index, offset within the page, etc) are used as input to a pipelined AES engine. The AES engine is keyed (e.g., it takes

advantage of the hardware private key), and produces a unique value as output referred to as a *one-time pad* (OTP). This OTP can be XOR-ed with the plain-text data to produce a cipher-text, and the cipher-text can be XOR-ed with the OTP to reproduce the plain-text. On a dirty writeback from the LLC (i.e., an update to the value of the data in memory), the counter associated with the data is incremented. As such, for each unique data value at each address, there is a unique OTP produced for encryption and decryption.

In practice, overflow of these encryption counters is a real concern. If two known plain-text values are encrypted with the same key, the hardware private key used to generate the OTP can be reverse engineered by solving a set of linear equations with a set of common variables. Such a case is vulnerable to an attacker who, knowing that overflows result in such collisions, will try to reverse engineer the hardware private key by repeatedly writing to an address (e.g., `for (;;) clflush 0xDEADBEEF;`) and leave the system highly vulnerable. As a result, on the overflow of a particular counter, the hardware private key needs to be re-derived. The outputted OTP from the AES engine will be different than the previous time the encryption counter was in this state. Re-deriving the hardware private key is a costly operation because it means that no OTP can be re-created. This means that all data in memory needs to be re-encrypted with the new hardware private key.

Reducing overflows in encryption counters is a critical problem to reduce the likelihood of total memory re-encryption. To address this problem, Yan, et. al. proposed using split counter-mode encryption [99]. Most modern works take advantage of this scheme. The work defines the sizes of "data" and describes a system of "major" and "minor" counters associated with each data. In particular, each 64B block of data (a word) has a unique minor counter (7 bits) and each 4kB of data (a page) has its own major counter (8 bytes). Put another way, there are 64 minor counters per page, with each word having its own unique counter. The page itself also gets its own major counter.

Both the major and minor counters are used as input to the AES engine in the OTP generation. Now, on a data write, the minor counter associated with that word is incremented. If the minor counter overflows, the major counter is incremented and all other minor counters are also reset to zero. Incrementing the major counter in this design has a similar effect as changing the hardware private key in the previously described example. That is, the OTP that was used to encrypt a value can no longer be generated as the major counter has changed. As such, each of the words in a page need to be re-encrypted each time the major counter is incremented so that the OTP used for encryption can be re-generated for decryption. Note, the minor counter for each of these OTP generations will always be zero, as each minor counter is reset on a major counter increment. An overflow of a major counter will result in needing to re-derive the hardware private key, but this case is a lot less likely as it requires overflowing a minor counter 2^{64} times.

Arranging encryption counters in this way achieves several properties that are advantageous for performance. For one, arranging the 64 minor counters per page as 7-bit values with a single 64 bit major counter means that there are 64 contiguous bytes that represent all minor and major counters for all words within a page. Accesses within page are likely due to the one-to-one mapping of virtual to physical addresses, so caching all counters for this page is likely to pay dividends. Furthermore, this setup is extraordinarily spatially efficient. Suppose a word is repeatedly flushed to over and over again (e.g. `for (;;) clflush 0xDEADBEEF;`). The minor counter for this address will be incremented repeatedly in a loop. One of them will overflow the minor counter before the other, which resets the minor counter for *both* addresses. The major counter can only be incremented by one of the two hot addresses. In essence, this creates a 71 bit counter for each word within the page (7 bits in the minor counter and 64 bits in the major counter). The

real space occupied by encryption counters is still only 8 bits per 64B of data (64B major-minor counters per 4kB of data).

Utilizing CME also has several performance benefits. To decrypt some data, the encryption counter must also be fetched, but the fetch latency can be hidden by the data access latency as both values can be accessed in parallel. Once fetched, the counter values, the page index, and the offset of the data word within the page (along with some padding) are fed as input to the AES engine. This engine is pipelined, and can output a OTP per single cycle under parallel access. Finally, the XOR operation between the OTP and the cipher-text is intentionally fast to produce the plain-text.

Encryption in CME is more expensive, as it requires fetching the major and minor counter values from memory in advance of encryption so as to increment the counter appropriately. Once incremented, the new OTP is generated and XOR-ed with the plain-text before the produced cipher-text is flushed to memory. Furthermore, if the minor counter overflows on incrementation, each of the other 63 words in the page need to be re-encrypted with the new major and minor counter (note, the new minor counter will be 0).

2.2.1 Prior Art Relating to CME

For about as long as CME has been around, there has been much art that has gone into the process of accelerating it at the hardware level. In particular, there are a few features of CME towards which the state-of-the-art has given attention. For one, an inefficiency of CME comes from the fact that OTP computation cannot occur until the major and minor counters associated with a data word are known (i.e., the process requires a memory fetch). In encryption, this fetch can be done in parallel with the fetch of the data, but even after the fetch there is a latency delay in returning data to the processor side until after the data is decrypted. Furthermore, counter overflows are significantly more expensive than typical operations. For the most part, their cost can be ammortized, but significant efforts have gone into how to reduce the likelihood of an overflow occurring. Finally, encryption counters incur significant spatial overhead. To implement CME, it requires 64B of metadata per every 4kB of data, which is significant.

In general, performance optimizations, reducing the likelihood of overflow, and reducing spatial overhead of CME can be classified into two categories: (1) improving CME by accelerating the procedure; and (2) improving CME by modifying the layout of encryption counters. Furthermore, as memory architectures develop and new emerging technologies gain traction, the literature adapts to the new requirements held by these technologies. For example, non-volatile main memories have implications on how and when counters may be cached on-chip. In this section, I will summarize these works.

TODO: finish this description of prior art relating to CME.

2.3 Integrity

Part II

Secure Memory Performance

Chapter 3

Memoized Counters

3.1 Introduction

With the growing use of remote services for computation on personal data, the issue of providing security and privacy has become a hot topic for some time now. When clients offload sensitive information to a remote machine, they do it in trust that they are protected from several attacks orchestrated by an untrusted OS or cloud administrators [53]. While subject to remote computation, some level of protection must be employed to compute sensitive data such as encryption keys, genetic information, blockchain transactions, etc.

In most scenarios today, this protection is guaranteed through secure computation solutions like Intel SGX that make use of *secure memory* [67]. Secure memory is defined by a protocol that makes use of a Bonsai Merkle Tree (Bonsai MT) [79]. This is a tree of hashes that is built on top of encryption-counters (to implement counter-mode encryption), and is coupled with data message authentication codes (MACs), which are hashes of data. To authenticate data, its counter is fetched and has its integrity verified against the tree, and the hashed data with its counter is compared against its MAC. However, secure memory has two fundamental limitations: (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 a lot of work towards resolving (1) [21, 12, 44, 57, 107], there has been a lot less work towards resolving (2) [93, 80, 99].

To alleviate this problem, we propose the *Baobab Merkle Tree*. The Baobab Merkle Tree takes advantage of the observation that many counters in memory have the same value. Given this, we propose an alternative protocol where encryption counter values are memoized in an on-chip table. In memory, only the index into the memoization table where the counter is located needs to be stored, which occupies $2 - 4X$ less space than the counter values themselves. As such, the Baobab Merkle Tree reduces the spatial overhead of the integrity tree by 2-4X. 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 Bonsai Merkle Tree equivalent.

In this paper, we present the following contributions:

1. We propose the Baobab Merkle Tree, which memoizes encryption counters in an on-chip table, decreasing the spatial overhead of the integrity tree by $2 - 4X$.

2. We define a technique to memoize encryption counters on-chip.
3. We model and evaluate the Baobab Merkle Tree in gem5 [64], and discuss the implications of its design trade-offs.

3.2 Background

3.2.1 Threat Model

We assume a well understood threat model where an attacker has physical access to the device. The attacker can snoop and/or modify data while it is in transport and stored in memory. We assume the processor chip is within the trusted computing base and data within the chip cannot be tampered.

3.2.2 Secure Memory

Data is encrypted with counter-mode encryption [79], where each data has a unique counter value that provides spatially and temporally unique encryption keys. The integrity of data is preserved by storing a keyed hash of that data and counter in memory (i.e., data MAC). When data is loaded on-chip, its hash (with its counter) is computed and compared against the MAC value. However, the MAC alone is not sufficient to protect data integrity against replay attacks, where the attacker corrupts the data, MAC, and counter to a prior value. At verification time without any additional integrity verification components, the system will succeed in verifying the integrity of the data even though it is not correct.

The Bonsai Merkle Tree protects the integrity of the counters and prevents replay attacks. It is a tree of hashes in which the root of the tree is stored on-chip (i.e., is trusted) and built on top of the encryption counters. To verify a counter with the tree, its hash is computed and compared against the stored value until a trusted value is reached, thereby preventing the reversion of a counter. On-chip metadata caches can be used to further optimize this approach. By caching recently accessed nodes on-chip, the authentication process is shortened as it can stop as soon as a node is found in the metadata cache—as this cache resides on-chip within the trusted boundary.

3.2.3 Secure Memory Optimizations

The notion of using memoization for accelerating the secure memory protocol is not novel. Recent art [96] works from a similar observation – many counters have similar values. However, their work, is concerned with redundant AES computation on these addresses from a performance perspective rather than on redundant storage. In fact, much of the attention in the state-of-the-art has been focused on reducing the performance overheads of secure memory [21, 12, 44, 57, 107], but there are far fewer approaches in which the spatial overhead of secure memory metadata is addressed.

Work in Synergy [81] describes a system by which the performance to access the data MACs is improved by storing these values in the ECC chips. This design eliminates an additional memory access to fetch the data MAC from memory — it is fetched at the same time as the data itself. Seeing as the target use case of secure memory is for highly secure systems, it is reasonable to classify benign hardware failures that lead to bit-flips in the same way as a breach of security.

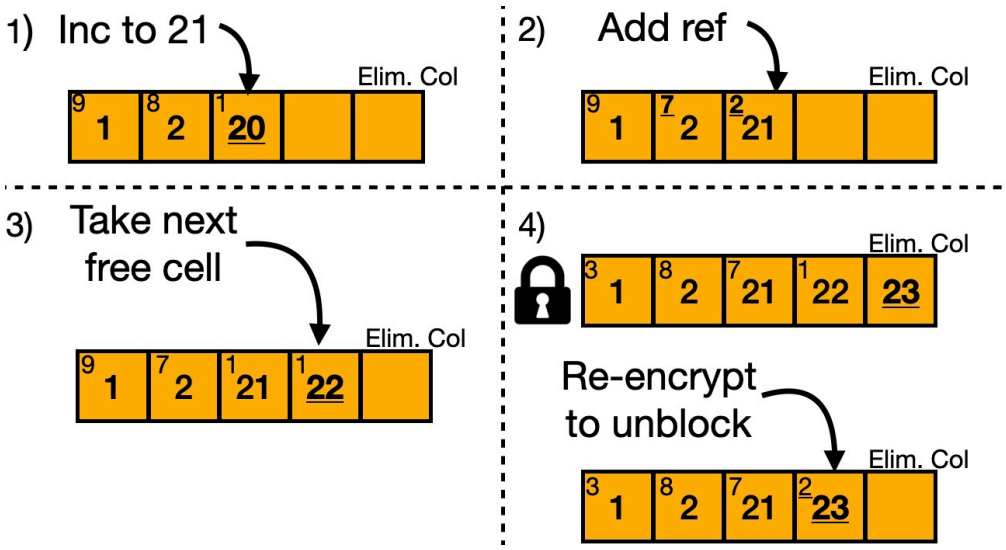


Figure 3.1: Incrementing counters using the elimination column.

Other prior work that focuses on spatial overhead is VAULT [93]. VAULT implements variable arities in BMTs to reduce the overall size of the integrity tree itself. In VAULT, the likelihood of a hash collision increases as the arity increases, but its procedure of hashing and re-hashing reduces the likelihood of several collisions. VAULT suffers from worse performance overhead compared to the baseline. In contrast, our Baobab Merkle Tree does not impact performance at all.

An alternative approach to improving the spatial overhead of the BMT is to dynamically adjust the arity of the tree on demand as it is done in Morphable Counters [80]. However, the key drawback of this approach is the frequent counter overflows and subsequent re-encryption that counteracts the spatial overhead savings.

3.3 Design

The Baobab Merkle Tree is a modification of the traditional Bonsai Merkle Tree 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*). A given 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. Critically, the total number of cells in the memoization table is dramatically smaller than the number of data blocks — the indices allow blocks to share counter values.

3.3.1 The Memoization Table

The memoization table is a fixed size buffer stored on-chip. 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.

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. 3.3.3, the Baobab system needs to consider the

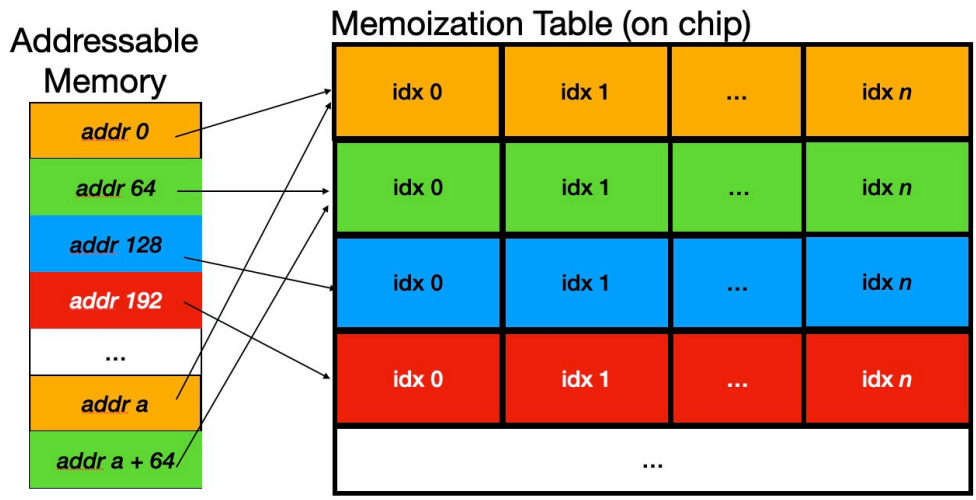


Figure 3.2: Memory assignment from address to memoization table row.

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" [85], commonly used for reference counting. For example, suppose we assume a 60 bit counter and 4 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.

3.3.2 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 columns per memoization table row. The physical address of the data is used to determine where this column index is stored, and the corresponding value at that address determines the column in the memoization table where the counter for en/decryption is stored.

3.3.3 Incrementing Counters

Incrementing a counter in the memoization table depends on its associated row's state and reference count. In particular, there are four types of increment in the memoization table: (1) in-place increment (2) next-cell increment, (3) free-cell increment, and (4) blocking increment. We use Fig. 3.1 to demonstrate each case.

In-place increment occurs when the block that requires incrementing the counter is the only block using that cell (Fig. 3.1, part 1). If the current cell holds the largest counter value in the row, or if the counter value in the current cell 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, the data can be written to memory with

limited additional overhead for secure memory relative to standard Bonsai Merkle Tree implementations.

Next-cell increment (part 2) occurs when the required cell is not the largest value in the row and the cell is not used exclusively or the next greatest counter is only one value larger than the current counter. In this case, the data block needs to now use the index of the next greatest counter in the row. As such, the index stored in the Baobab Merkle Tree has changed, and the tree needs to be updated. In terms of memory operations, this case exhibits similar behavior to a standard write in a Bonsai Merkle Tree. 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.

Free-cell increment (part 3) occurs when data uses the cell with the highest counter, but that counter is not held exclusively (i.e., the memoization table has a cell with a reference count of zero). In this case, the increment uses the free cell, filling it with the value of the incremented prior counter.

Blocking increment (part 4) occurs when the data uses the cell with the highest counter, but that counter is not used exclusively, and the memoization table has no free cells available to reuse. In this case, the system reserves the last column of the memoization table row as the "elimination column." Suppose, after some time, the row takes the state of the upper row in Fig. 3.1 part 4. In order to increment from 22 (i.e., next cell increment), the elimination column is filled. This locks further authentications to the row to avoid conflicts. Then, in the lower row, unblocking is achieved by scanning for the least referenced cell in the row and reencrypting those data with the new counter value created in the elimination column (i.e., reencrypted with 23). The 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, we need to perform a *reverse mapping* from counters to data to check which data points to that column and needs to be reencrypted. To ensure that there is adequate hardware, while reencryption is happening we block all authentications that require this memoization row.

3.3.4 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, each row assignment works from a heuristic to increase the likelihood of in-place increment and decrease the likelihood of needing a blocking increment.

We work 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 rows. If no physical locality is observed, blocks will need to increment counters at similar but slightly different rates, which will occupy more columns per row. The proposed row assignment is to "stripe" the memory to memoization table row, as per Figure 3.2.

3.3.5 Security Implications

In order to uphold secure memory semantics, Bonsai Merkle Trees protect the integrity of encryption counters and use data MACs to ensure that data hasn't been corrupted [99]. The intuition is that only the untampered encryption counter can produce the decryption key that decrypts the data to plaintext 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 Bonsai Merkle Tree would detect tampering or replaying of encryption counters in memory.

3.4 Evaluation

3.4.1 Methodology

We implement the Baobab Merkle Tree as an extension to gem5 [64], a full system processor simulator. We configure a four-core simulation 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). We 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. We run two baseline approaches, one with a comparable metadata cache size to the Baobab Merkle Tree (i.e., 32kB) and one with a comparable on-chip resource size (i.e., 256kB metadata cache). We 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, we prefill the memoization table state. The prefilled contents are collected from memory traces of each of the SPEC 2017 CPU benchmarks [14] run back-to-back while modeling what the table state would be offline from the simulation.

3.4.2 Spatial Overhead

The spatial overhead of Merkle Trees in secure memory scales proportionally to the overall memory size. Table 3.1 shows the amount of reserved memory space required to store the integrity tree. The size of the Baobab Merkle Tree is shown under *Baobab MT*, while the size of the size of Bonsai Merkle Trees is shown under *Bonsai MT*. The fact that we can protect and authenticate twice as much data per leaf in the Baobab Merkle Tree versus the Bonsai Merkle Tree means that the Baobab Merkle Tree *requires half as much space* in memory as the Bonsai Merkle Tree.

The Baobab Merkle Tree size strictly depends on the number of cells within a memoization table row. If, for example, we store 4 columns per row, then only 2 bits are required to track the index into the memoization table row, and thus the Baobab Merkle Tree has a spatial reduction of $4X$ rather than $2X$ (256-ary versus 128-ary leaf level). However, we opted for 16 columns per row in our approach in order to limit the number of blocking cases. 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.

3.4.3 Runtime Evaluation

We use the SPEC 2017 CPU benchmarks [14] to evaluate the overhead of the Baobab Merkle Tree over a Bonsai Merkle Tree. Fig. 4.5 shows that, on average, the Baobab Merkle Tree implementation does not impact performance; it has an average performance benefit of less than one percent. As per [96], the latency to update an memoization table entry is negligible relative to the memory access latency. Furthermore, we find that the metadata cache hit rates are very high in the baseline approaches. Given these factors, the Baobab Merkle Tree has no significant overhead nor speedup relative to the baseline secure memory model.

The Baobab Merkle Tree has a significant reduction in metadata cache misses relative to the Bonsai Merkle Tree baseline, even though more on-chip space is used by the metadata cache. Fig. 3.4 shows the number of overall metadata cache misses 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 lower cache miss rates.

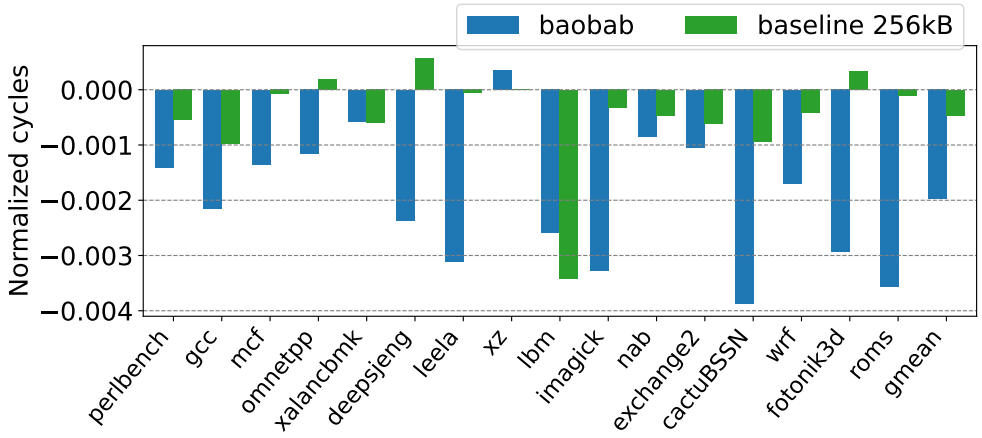


Figure 3.3: Normalized cycles in the SPEC 2017 CPU benchmarks.

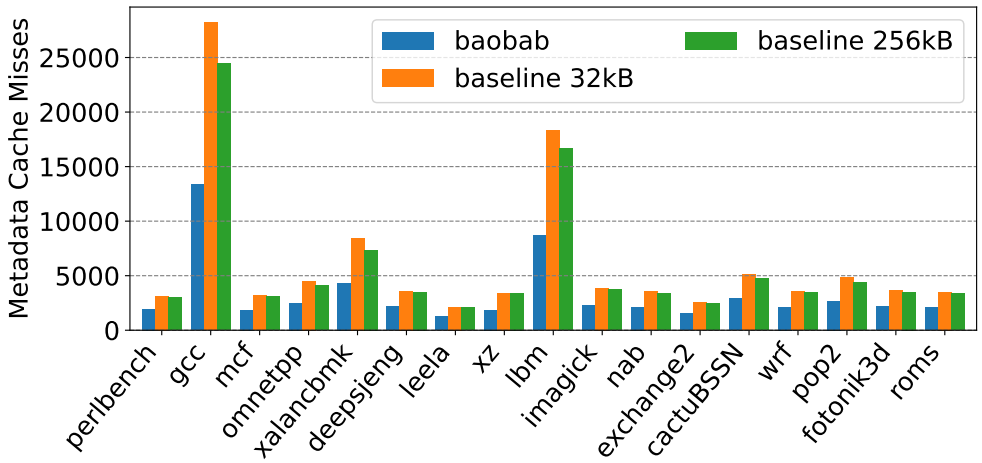


Figure 3.4: Metadata cache misses in SPEC 2017 CPU benchmarks.

3.5 Conclusion

In this paper we present the Baobab Merkle Tree. We show that, because indices require fewer bits than the counters themselves, the Baobab Merkle Tree reduces the spatial overhead of the integrity tree by 2-4X. 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.

Table 3.1: Description of the spatial trade-offs in the Baobab Merkle Tree for varying memory sizes.

	Blks/Row	Baobab MT	Bonsai MT
256G	16M	**2.5GB**	5GB
1TB	67M	**9.5GB**	19GB
8TB	536M	**78.5GB**	157GB

Chapter 4

Crash Recovery

4.1 Introduction

Traditional systems with volatile memory technology suffer from active and passive physical attacks. These have been thoroughly investigated over the past two decades. However, since volatile memory systems lose their state when power is disconnected, prior work did not have to address the data remanence problem [90]. Storage class memory (SCM) systems use non-volatile technologies as the main memory, so data will remain intact even after power is disconnected. Furthermore, SCM systems are vulnerable to a new set of physical attacks since this non-volatility gives attackers a larger window of opportunity for performing splicing, spoofing, and replay attacks. Thus, SCM is a natural target for secure memory guarantees and protections.

Standard approaches for providing secure memory for volatile systems use counter-mode-encryption to protect confidentiality and Bonsai Merkle Trees (BMTs) with keyed hash message authentication codes (HMACs) to protect data integrity [101, 69, 79, 91]. The key insight behind these approaches is that the chip is *too small* for an attacker to inspect or inject malicious content and thus everything on-chip is trusted and everything off-chip is not. Any data request that needs to leave the trusted boundary goes through the integrity verification process (a traversal of the BMT) and decryption. The root of the BMT is always trusted as it is stored on-chip (within the trusted boundary). Any authentication failure would deem that the data has been corrupted, and could maliciously taint the execution in order to hijack control-flow [15, 75, 1], escalate process privileges [19, 31], or any number of other memory corruption-based attacks [84, 83, 68, 27]. When an integrity verification fails, prior work assumes the system can simply reboot. With SCM systems, rebooting is not an option, as SCM semantics imply that the corrupted values will remain in memory throughout the reboot, and can still maliciously impact a victim's execution. Furthermore, many cloud service providers rely on instantaneous recovery to maintain quality of service agreements [73, 17, 9].

Secure memory systems for volatile memory cannot simply be retrofitted to work with an SCM system as their lack of *crash consistency* means that they cannot validate integrity across unexpected reboots. An SCM system usually relies on a persistence model to enforce the persistence of particular data. However, the secure memory system requires metadata to be persisted along with the data for recovery after power loss. For example, if a block at time t (p_t) is persisted to memory, it needs to first be encrypted with a counter (C_t) which is stored in memory, then its hash (H_t) needs to be computed and also placed in memory, and finally the integrity tree (T_t) needs to be updated to reflect the new encryption counter value. If the

metadata (C_t, H_t, T_t) is not persisted at the same time as the data (p_t), then if a power outage were to occur, the persisted data (p_t) would not satisfy the integrity verification requirement as the metadata would not reflect the persisted state (C_{t-1}, H_{t-1}, T_{t-1}). Maintaining crash consistent security metadata along with the associated data as part of the persistence model is the essential challenge for secure SCM.

To provide crash consistency, 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, in that it can lead to steep performance overheads (up to $25X$) at runtime.

An alternative approach, dubbed *leaf metadata persistence* [103], 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 scalably 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 has worked to achieve a "best of both worlds" protocol within this trade-off space. However, while these protocols negotiate performance and recovery, they introduce a third-component to the trade-off space by not considering the area overhead of their approaches. In particular, they assume larger metadata caches than in Intel SGX (64kB) [40]. For example, several works assume large metadata caching structures [106, 20, 58, 37, 38, 47] or assume physically large, NVM-based devices on-chip [32].

However, an implicit design goal of secure memory is to *minimize* the amount of on-chip area required to realize the protocol, as on-chip space is more optimally used for data storage. Space from a large metadata cache can be better used to store data in a larger LLC — a larger LLC reduces the number of instructions that use untrusted memory and thereby avoids additional integrity checks [57].

In this paper, we propose *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 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. We leverage 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,

[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."

the maximum stale data size (defined by the level at which the subtree root is placed). In this paper, we demonstrate that this insight holds true for several applications with varying characteristics. For adversarial cases, we turn 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.

We make the following contributions:

- We present AMNT, a novel and efficient mechanism to persist security metadata alongside data.
- We introduce 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.
- We describe a low-overhead recovery mechanism for SCM.
- We show how AMNT reduces volatile on-chip space by $49X$ or non-volatile on-chip space by $32X$ versus the state-of-the-art alternatives.
- We demonstrate how a system administrator can bound the recovery time using our proposed approach to achieve desired performance goals.

4.2 Background

Much research has gone into designing efficient techniques for ensuring secure traditional (i.e., volatile) main memory. However, new security challenges arise with storage class memory, which also affect runtime performance and recovery.

4.2.1 Traditional Secure Main Memory

A traditional secure memory system provides both confidentiality and integrity verification for data in memory. In order to provide confidentiality of data in memory, state-of-the-art integrity protected memory schemes rely on *counter-mode encryption* (CME), which uses a counter and address as input to an AES engine that produces a one-time pad. To encrypt plaintext data, the one-time pad is XOR'ed with the data to produce the ciphertext. To decrypt that data, the same one-time pad is XOR'ed with the ciphertext to produce the plaintext. To balance cache efficiency and storage overhead, each 64B block has a unique minor counter (7 bits), and each 4KB page has a unique major counter (8 bytes). Together, the major and minor counters ensure that CME provides each block a spatially and temporally unique encryption key while minimizing the cost of a counter overflow.

Integrity protection is accomplished by computing and storing a hash on a data write. To provide strong security guarantees the hash used is a keyed hash message authentication code ($HMAC$). When data is fetched from memory, the hardware can confirm that it has not been corrupted by comparing a newly computed HMAC against the previously stored HMAC. In the event that the stored HMAC does not match the newly computed one, the processor simply reboots to restart from a "safe" state. The HMAC alone cannot provide full integrity protection due to its inability to detect replay attacks. For example, an attacker may observe old data and HMAC, and replace legitimate values with stale values that still verify.

To provide protection from replay attacks, state-of-the-art secure memory systems use a *Bonsai Merkle Tree* (BMT) [79], which is a modified Merkle Tree (a tree of hashes) that protect the integrity of the encryption counters [69, 79] (depicted in Figure 4.1). The BMT is composed of hashes of the counters at the leaves. These hashes are hashed together to produce the node of the next level (the parent node). This process is repeated recursively until a single node remains (i.e., the root of the tree). To establish the root of trust, the root of the tree must always be stored on-chip, as this is the trusted boundary. The BMT is an efficient way of constructing and storing an integrity tree as it reduces the tree's memory requirement [7, 36, 8]). Furthermore, the BMT root is small and unique to the state of the entire underlying memory state, so it provides a verification mechanism with very little on-chip area overhead.

In order to verify an untrusted block of data coming from memory against the BMT, the ancestral path of the data through the tree (i.e., counter, tree nodes, root) must be fetched, and the hashes must be recomputed and compared against the stored values. If each hash in the process matches up to the root (trusted value), then the integrity of the data is authenticated. Similarly, updating a data value requires updating the values of nodes in the ancestral path in the BMT to reflect the state of the new data.

To reduce the integrity verification latency, most prior work assumes that security metadata (integrity tree, encryption counters and data HMACs) is kept in on-chip caches [34, 57, 56, 80, 59, 78, 81, 87, 86, 91, 92, 99, 101, 94, 98]. Doing so introduces a two-fold optimization. First, like data caches, recently accessed metadata can be re-accessed with lower access latency. Second, nodes cached on-chip are members of the trusted compute base, so they too can act as roots of trust, reducing the integrity verification path.

Bonsai Merkle Trees come in two forms: General BMTs and SGX-style BMTs. General BMTs describe a BMT structure where nodes are the concatenated hashes of their children (like that in Figure 4.1). SGX-style trees have counters embedded in each node throughout the tree, concatenated with a hash of its parent. In this form the hash is computed using the Galois-counter hashing mode [36]. The experiments in this work assume the General BMT format. However, the proposed protocol can be used in an SGX-style BMT with small modifications to take advantage of underlying protocols that target this format, such as Anubis [106] and Osiris [103].

4.2.2 Secure SCM

Storage class memory (SCM) systems introduce a new challenge to the consistency of data, namely, data persistence. 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 the non-volatile properties. Various software libraries (e.g. [18, 49, 77, 95, 22]) and hardware extensions (e.g. [105, 71]) provide the programmer sufficient control to avoid data inconsistency problems and ensure application data is crash consistent and usable after a power failure.

Like application data, security metadata (i.e., HMACs, counters, and the BMT) must also be crash consistent. Otherwise, integrity verification would not be possible. However, unlike in application data, secure memory metadata is not accessible by the application, so persistence of secure memory metadata must be enforced by the hardware. 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

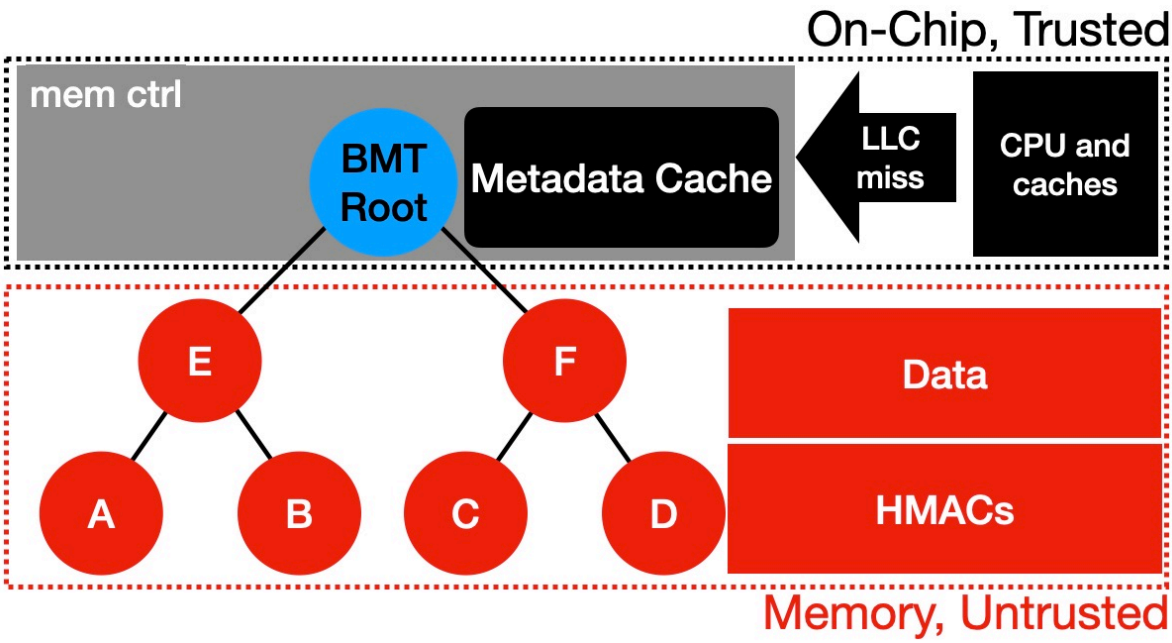


Figure 4.1: Bonsai Merkle Tree (BMT). Inner tree nodes are the concatenation of the child hashes. through a crash.

4.2.3 Metadata Persistence Strategies

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. Unfortunately, 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 in main memory, and updates to the BMT (and BMT root) must be atomic with the data write. 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 is a lazy technique for improved runtime performance but dramatically increases recovery time. In this strategy, only the BMT leaf (i.e., counter) update is done atomically with a data write. The rest of the security metadata is updated lazily (as done in the volatile secure memory systems). 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.

Prior state-of-the-art work Osiris [103] 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

(a) Trace of *perlbench*.

(b) Trace of *lbm*.

Figure 4.2: Memory accesses per address in different SPEC CPU 2017 benchmarks.

data updates to ensure that they can never be more stale than the stop-loss frequency. 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. As a result of the persistence relaxation, full BMT recovery in Osiris is slower than in leaf metadata persistence.

Another state-of-the-art technique, Anubis [106], takes an approach that tracks the addresses currently residing in the metadata cache — only these addresses need to be recomputed at recovery time. It tracks this information by "shadowing" these addresses in a new structure in memory, called the "shadow table". This shadow table tracks updates to cached metadata nodes in order to have a precise state of stale metadata. Instead of performing whole tree reconstruction, this protocol corrects only the stale values in memory at recovery time. Seeing as the shadow table of addresses resides in memory, it must have its integrity protected via a secondary "shadow Merkle tree." The shadow Merkle tree must be updated whenever the shadow table is updated, which happens on a metadata cache miss (i.e., a new value is placed in the metadata cache). This technique makes recovery a function of metadata cache size rather than memory size, but it is also a movement away from reducing the spatial overhead of secure memory [93, 80, 79].

On the other side of the spectrum, Bonsai Merkle Forest [32] reduces the overhead of strict persistence by storing frequently accessed subtree roots in a non-volatile on-chip cache. In order to determine which nodes qualify, counters are added to the the volatile metadata cache lines, and nodes in the persistent root set (i.e., non-volatile metadata cache) are either "pruned" or "merged" depending on whether the hotness of the node increases or decreases.

4.3 Motivation

The notion of spatial locality is one that is well understood at the software level, and spatial locality in the virtual address space is an important consideration for application performance. However, few applications consider how locality among virtual addresses could continue across pages to physical addresses. As the BMT is organized on physical addresses, we explored *physical address* locality in a variety of benchmarks and found, somewhat surprisingly, that many applications exhibit spatial locality on physical addresses.

To see the precise in-memory footprint of different applications, we collected memory traces from each of the SPEC 2017 CPU [14] benchmarks, along with several combinations of benchmark workloads to emulate the impact of multiprogram workloads. The memory traces are from dumping addresses accessed in memory (i.e., LLC misses and writebacks) from a full system mode simulation in gem5 [13], which runs on an Ubuntu 18.04 simulated disk image and with Linux 4.14. We simulate for 10 billion instructions, and other simulation details are described in Sec. 4.7.

In Fig. 4.2, we show memory traces of two workloads that exhibit different characteristics. *Perlbench* (Fig. 4.2a) is not particularly memory intensive (3 MPKI), and so a significant portion of its instructions executed are in the OS. It shows two peaks – one in memory for the OS address space, and one at the application's address space. On the other hand, *lbm* (Fig. 4.2b) is much more memory intensive (25 MPKI), and a larger percentage of its execution is spent accessing the memory regions. In this benchmark, there is a large number of contiguous addresses accessed at a very high rate in memory. These figures demonstrate that, to a certain extent, applications can assume that contiguous virtual addresses may largely be in contiguous physical spaces. As predominant applications occupy the majority of execution within a device, they are able to acquire the associated physical resources contiguously. As previously expressed, applications across many domains take advantage of large contiguous virtual addresses, such as the stack. Although the mapping of virtual to physical addresses can only theoretically be expressed as contiguous within a page, our empirical study of *physical addresses* accessed in memory demonstrates that, in practice, this assumption extends to the physical address space as well. This finding makes sense—Linux's buddy allocator merges contiguous pages on reclamation, and as such they exhibit good physical locality on the next intense period of allocations. Thus, we can leverage this behavior when designing AMNT.

4.4 Threat Model

We assume a well understood threat model in which on-chip values are trusted and everything else is untrusted [91, 60]. The off-chip components (*i.e.* memory, disk, I/O) are vulnerable to passive (snooping) and active (splicing and replay) physical attacks, from both legitimate and illegitimate users. The threat model also includes remote users who can perform privilege escalation attacks remotely or an untrusted operating system, giving them privileged access to any region in memory. Our proposed work is focused on protecting in-memory data in a system that uses a non-volatile device as the main memory. The attacker is assumed to have both physical access to the system and is able to run any legal program to exploit potential off-chip physical device vulnerabilities. Similar to prior work [106, 103, 5, 33, 32], protection against information leakage via side-channel attacks is out-of-scope for our proposed defense mechanism.

4.5 A Midsummer Night's Tree

Our 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. As a result, there is more room for larger on-chip caches (i.e., LLC) which will reduce the number of instructions that use the secure memory protocol. AMNT achieves its goals by using multiple metadata persistence strategies within the same BMT [79]. As discussed in Sec. 4.3, 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. We also propose an optional mechanism to further optimize AMNT, which we call AMNT++, which adds a lightweight modification in the physical page allocator to further improve the design's performance.

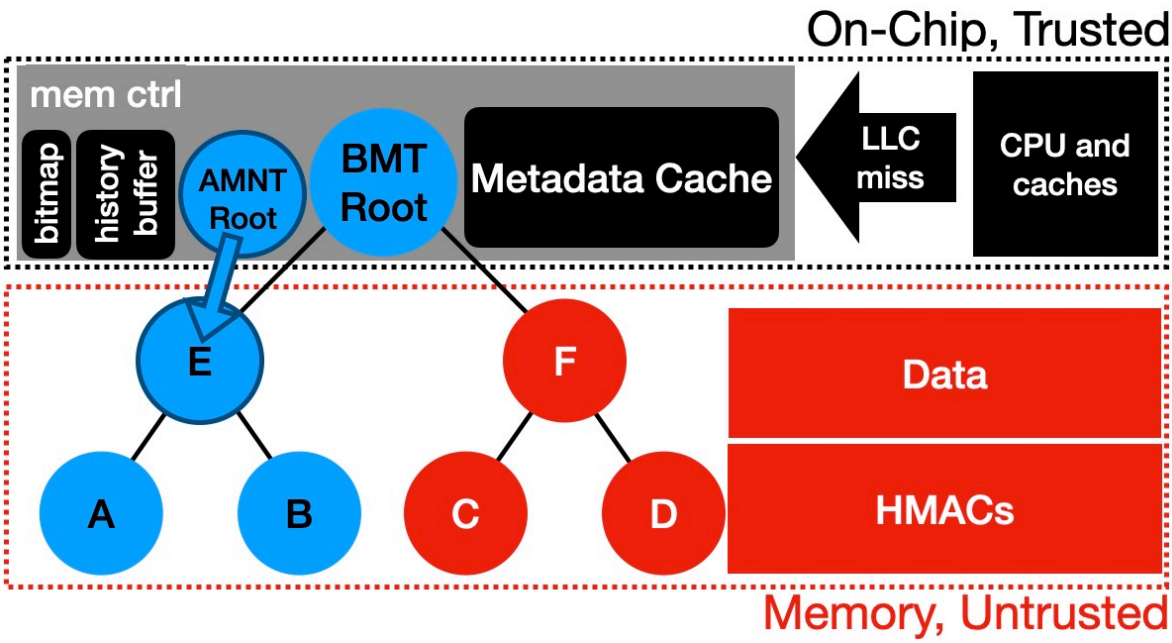


Figure 4.3: A Midsummer Night's Tree. Red nodes implement strict persistence. Blue nodes implement leaf persistence.

4.5.1 "Tree Within a Tree"

AMNT is a metadata persistence protocol that tracks hot regions of physical memory within a *subtree* of the underlying BMT. The subtree implements a leaf metadata persistence strategy, where tree nodes assumed to be stale at the time of a system failure (blue nodes in Fig. 4.3). The rest of the BMT implements a strict metadata persistence strategy to minimize recovery time after a crash (red nodes in Fig. 4.3). Our design expects that a small percentage of memory will be accessed frequently, and the system prioritizes making those accesses as fast as possible. Implementing a strict metadata persistence strategy in the region outside of the subtree, while slow at runtime, will not occur often, thereby minimizing impact on overall performance and accelerating recovery time.

To implement the AMNT protocol, the BMT is split into the main tree abiding by strict metadata persistence semantics (slow runtime, fast recovery) and a subtree that abides by the leaf metadata persistence semantics (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. This BMT subtree root register enables fast write-through operations that are persistent and in the trusted domain. The rest of the subtree is persisted in SCM. Our approach makes data updates within the subtree much faster — the associated tree node update can take advantage of the metadata cache instead of going to main memory. 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 updated (and persisted) in memory.

Given the metadata persistence strategy, all values outside the subtree root in the BMT are up to date at the time of a crash. In order to recover the BMT, AMNT needs to only recompute the nodes inside the subtree; the recovery time depends on the size of the subtree which is, in turn, determined by the level of the subtree root. To let system administrators control recovery time, the level of the subtree root can be configured in the BIOS.

4.5.2 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 the most frequently accessed region in memory (i.e. the subtree root can move horizontally in the tree). Each node at this level protects a portion of memory, termed the *subtree region*. In order to efficiently determine the most frequently accessed subtree region, AMNT makes use of a lightweight history buffer.

Our history buffer has n entries and tracks the n most recent memory updates. Each entry has a subtree ID (identified by the index of the node within the subtree level) and a $log_2 n$ counter. On an access to some data within a subtree, the corresponding subtree index is updated in the history buffer by 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). To minimize runtime overhead, the history buffer is *not* fully sorted, but the head element is the maximum. After n data updates to memory, 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.

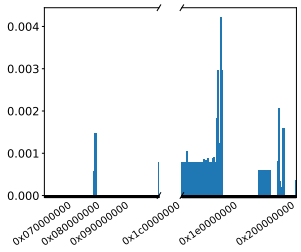
When transitioning from subtree T to T', all inner integrity nodes of T must be persisted before T' can implement the leaf persistence protocol in order to preserve the crash consistency and security guarantees. Note that the only ancestral paths from subtree T that need to be written to memory are those originating from modified (dirty) data.

To reduce the time to transition from one T to T', AMNT tracks "dirty paths" in subtree T, as only these paths need to be persisted. We track dirty paths by leveraging a small on-chip bitmap, where every bit in the bit map corresponds to a cacheline in the metadata cache. If the cacheline refers to metadata that falls within the subtree region belonging to T and that cacheline is dirtied, then the bit is set and the path from that line to the root needs to be persisted directly. Fortunately, if multiple cachelines correspond to a single path to the root, the updates along that ancestral path can be buffered [33]. The "dirty path" bitmap is very small. In a 64kB metadata cache, the bitmap is 128 bits. If there is a crash while the subtree T is being persisted, then it is possible that the inner integrity nodes of T have not been persisted. However, at this point T is still the subtree root tracked by AMNT, so all of T will be reconstructed at recovery time.

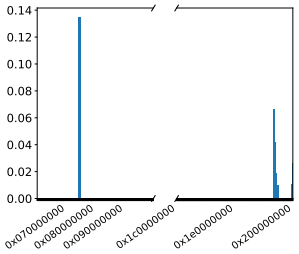
The history buffer (and corresponding bitmap) is an efficient and lightweight method to track the most frequently used regions of memory to select the best subtree root. Each entry in the history buffer requires at most $log_2 n$ bits for the region's index and an additional $log_2 n$ bits for the counter, resulting in $n * (2 log_2 n)$ additional bits. For a subtree at level three (64 possible subtree regions), this is 896 bits of additional on-chip area overhead (768 bits for the history buffer and 128 bits for the dirty path bitmap). The logic to update the buffer is a simple add and comparator that updates the head of the buffer based on if the counter is larger than the last head. In the event of a tie, the current subtree root stays at the head of the buffer to avoid a subtree movement.

4.6 AMNT++

In order to further optimize AMNT, we explored an optional *hardware-software co-design* mechanism to improve hit rates in the fast subtree. Consolidating frequent memory accesses into a single subtree can be an important performance optimization, which we attempt to maximize through lightweight modifications



(a) Trace of multiprogram.



(b) Multiprogram, modified OS.

Figure 4.4: Memory accesses per address in *cactuBSSN* and *nab* for unmodified and modified physical page allocators.

to the OS memory management module and its physical page allocator. Using this optional OS modification increases the efficacy of the underlying hardware while still keeping the hardware simple.

4.6.1 Biased Physical Page Allocator

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 operating system (OS) over time. This 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, we modify the buddy allocation from the Linux operating system [62]. Our modified OS achieves this by reordering the free area to have the chunks within a contiguous 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 linked list at `free_area` 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 our modified version of the buddy allocator, we modify the linked list structure by prioritizing 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 and our modification then reorders the linked list 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 restructuring function first scans each linked list to count how many chunks fall under each 64MB region. When the OS finishes scanning the list, it selects the region with the greatest number of chunks (the largest free contiguous physical area) and then moves all the ordered chunks for that region to the front of the linked list in a separate linked list (not in the `free_area` struct). Once the OS is done with

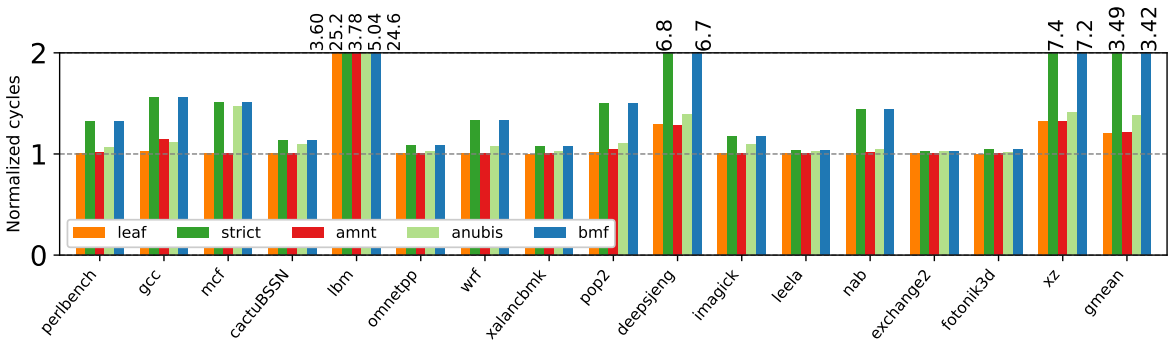


Figure 4.5: Runtime comparison of AMNT, Anubis, and BMF protocols for the SPEC 2017 CPU benchmarks normalized to writeback secure memory protocol. Lower is better.

the restructuring, the OS replaces the linked list with the new biased version. 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 the page allocation call).

4.6.2 Impact of Modified OS on Physical Locality

In a system with lots of processes being created and reclaimed, this means that the our one-program simulations demonstrate idealistic conditions. As such, our memory traces from Fig. 4.2 shows idealistic scenario. To create more noise in the system, we ran multiprogram workloads (described in more detail in Sec. 4.7), and show them in Fig. 4.4a.

Our modified OS increases the physical contiguity of hot regions of memory. Fig. 4.4b shows the memory trace of the *cactuBSSN* and *nab* multiprogram workload under the modified physical allocator, in which there were multiple spikes of hot regions in the unmodified OS. As demonstrated, even though the execution describes a multiprogram workload, we can see that the peaks of hot memory regions are much closer together than in the unmodified OS. Also of note is that a larger percentage of memory accesses go to the OS region of memory in order to re-organize the `free_area` structs. This shows that our modified physical page allocator increases the physical locality of application data. Thus, the overall contiguity of in-use physical addresses will be increased, and the performance of AMNT will improve through a higher subtree hit rate.

4.7 Evaluation

4.7.1 Methodology

We implement AMNT as an extension to gem5 [13], a cycle-accurate full-system simulator. We simulate a system with 8GB of phase-change memory (PCM) as the main memory, with latencies modeled after prior work on PCM [50, 43]. This configuration is similar to the one assumed in prior work on secure memory SCM [106, 5, 103]. We assume a specialized 64kB metadata cache for the security metadata as exists in Intel SGX [40]. The full configuration details are described in Table 4.1.

We evaluate AMNT against several state-of-the-art approaches in the BMT as implemented in [79] as well as two persistent protocols (leaf and strict) that represent the two extremes between runtime performance and recovery time. Our tests include:

Table 4.1: Evaluation framework configuration.

On-Chip Configuration	
Processor	4 cores, ARM ISA, out-of-order 1GHz clock, 1 thread/core
L1 cache	48kB icache, 32kB dcache 2-way set-associative LRU 2-cycle latency, 64B/block
L2 cache	512kB, 8-way set associative LRU 20-cycle latency, 64B/block
L3 cache	8MB, 64-way set associative LRU 32-cycle latency, 64B/block
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 [50], 391ns write [43]

1. **AMNT** our hardware implementation described in Sec. 4.5. All evaluation referring to "AMNT" refers this implementation.

2. **AMNT++** our AMNT design including the modified OS described in Sec. 4.6.

3. **Writeback (WB)**: Security metadata is only written-through to memory as it is evicted from the metadata cache. This cannot guarantee crash consistency. We consider this approach to be the baseline.

4. **Anubis** [106]: Anubis tracks all possibly stale metadata locations in the BMT, with each location protected through the implementation of a second, smaller BMT, designated here as the shadow Merkle Tree. In the event of a crash, only the BMT nodes tracked in the shadow Merkle Tree need to be updated, eliminating the need for a full-tree recovery.

5. **Bonsai Merkle Forest** (BMF) [32]: The Bonsai Merkle Forest is designed to reduce the write path for hot nodes. The protocol keeps the hottest nodes in a separate nonvolatile metadata cache to reduce the leaf-to-root path length. Our implementation of this protocol assumes a 4kB nonvolatile metadata cache and a 64kB metadata cache.

6. **Leaf**: This strategy guarantees that encryption counters (i.e., BMT leaves) are written through to memory. Inner BMT nodes are assumed to be stale at the time of recovery, and must be reconstructed. As such, this solution has the slowest recovery time.

7. **Strict**: All BMT nodes are explicitly written-through to main memory. No metadata is stale at the time of recovery. This solution has the fastest recovery time.

Anubis [106] assumes that the shadow table is tracked in a separate volatile on-chip cache to reduce the number of memory writes. Similarly, BMF [32] requires the utilization of several kilobytes of additional caching for the persistent root set. We include the shadow table in a distinct on-chip cache in our implmentation of Anubis, and we model the non-volatile metadata cache in BMF as a distinct on-chip cache. However, to normalize the hardware evaluation, we run each configuration with a 64kB metadata cache for the underlying secure memory protocol to remain consistent with Intel SGX [40].

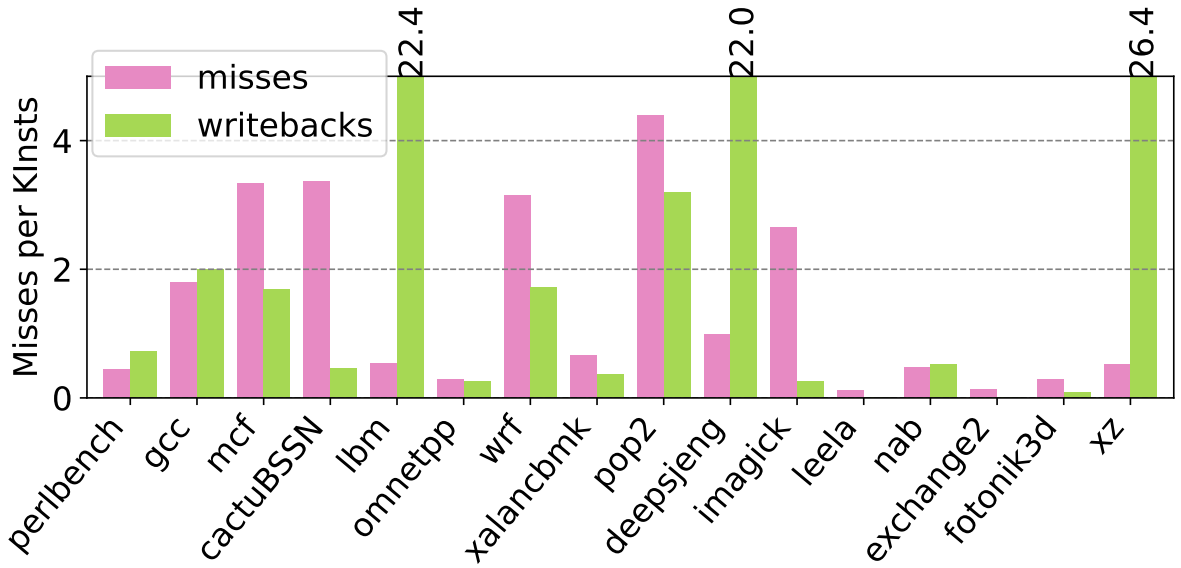


Figure 4.6: Misses in LLC per 1000 instruction for the SPEC CPU 2017 benchmarks.

4.7.2 The SPEC CPU 2017 Suite

We evaluate AMNT on the SPEC CPU 2017 benchmark suite [14]. We run the speed benchmarks with ref inputs, and we fast-forward to a region of interest as determined by SimPoint [76] in the benchmark before simulating 500 million instructions to sample the execution. This approach is consistent with prior work [106, 103, 5, 33, 12].

Figure 4.5 shows the normalized cycles of the SPEC CPU 2017 benchmarks over a WB 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 [106]. Compared to the naïve implementations of the two extreme persistency models, AMNT has a runtime overhead of less than 2% compared to leaf metadata persistence, and up to an 8X reduction in overhead relative to strict metadata persistence (shown in Figure 4.5).

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 an $7X$ overhead. To further emphasize this point, Figure 4.6 reports the number of last-level cache (LLC) behavior per thousand of instructions (MPKI) to show the memory intensity of the SPEC CPU 2017 benchmarks. 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 demonstrate larger overheads between insecure and secure configurations, as reads 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 WB in *cactuBSSN* and *mcf* because they are read-mostly memory-intensive benchmarks. Yet, Anubis and BMF both have significant overhead, because AMNT better optimizes metadata cache behavior.

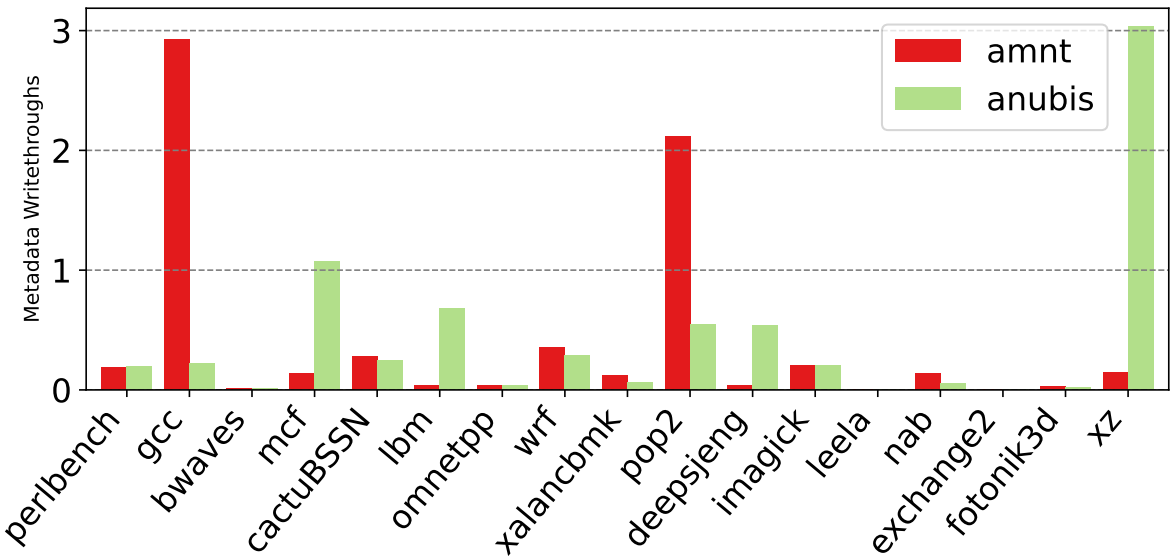


Figure 4.7: Number of metadata writethroughs for AMNT and Anubis per 1000 instructions.

AMNT's runtime performance versus Anubis can be measured by the number of write-through operations to secure memory metadata due to the crash-consistency model (shown on Figure 4.7). As expected, *xz*—a write-mostly workload with good subtree locality—has most operations benefiting from leaf metadata persistence in AMNT, and as a result has essentially the same runtime overhead as the leaf persistence setup. However, it inefficiently utilizes the metadata cache, and exhibits high runtime overhead in Anubis. Put another way, AMNT better optimizes for metadata cache misses than Anubis. Furthermore, due to its memory intensiveness, the overall performance benefit in AMNT is much greater. On the other hand, *gcc*, which has the highest subtree miss rate of the SPEC 2017 CPU suite, has only a 14% overhead in AMNT, which is comparable to Anubis due to the lack of memory intensity of the benchmark (less than 4 MPKI).

4.7.3 Memcached with YCSB

We use memcached [30], a key-value storage application developed by Facebook, to evaluate more realistic workloads that typically run on SCM devices. We run a client and server application on the same simulated machine, where the client connects to and accesses the server via the memcached client. Then, we query the server application from the client according to YCSB [25] workloads A, B, C, and write-only. In workload A, 50% of queries are reads and 50% are updates; in workload B, 95% of queries are reads and 5% are updates; in workload C, 100% of queries are reads; and in the write-only workload 100% of queries are set operations. In YCSB, keys are queried according to a zipfian distribution. Omitted YCSB workloads make use of the "scan" operation, which cannot be implemented in memcached without modifying the backend and is out of scope for this project. In our evaluation, we use a key space of 10,000 elements and we use values of size 8kB. We prefill the key space to 50% capacity, which results in a memcached server prefilled to 40MB.

Memcached does not exhibit as many memory operations as some of the more memory intensive SPEC benchmarks, as demonstrated by the lower number of miss per thousand instructions. Even with $2X$ and $4X$ larger key spaces with uniform distributions rather than zipfian, the overwhelming majority of operations occur in the software layer. As such, the differences in performance in Figure 4.8 are less significant. This is largely due to the fact that the memcached server performs significant statistical logging, and the memcached

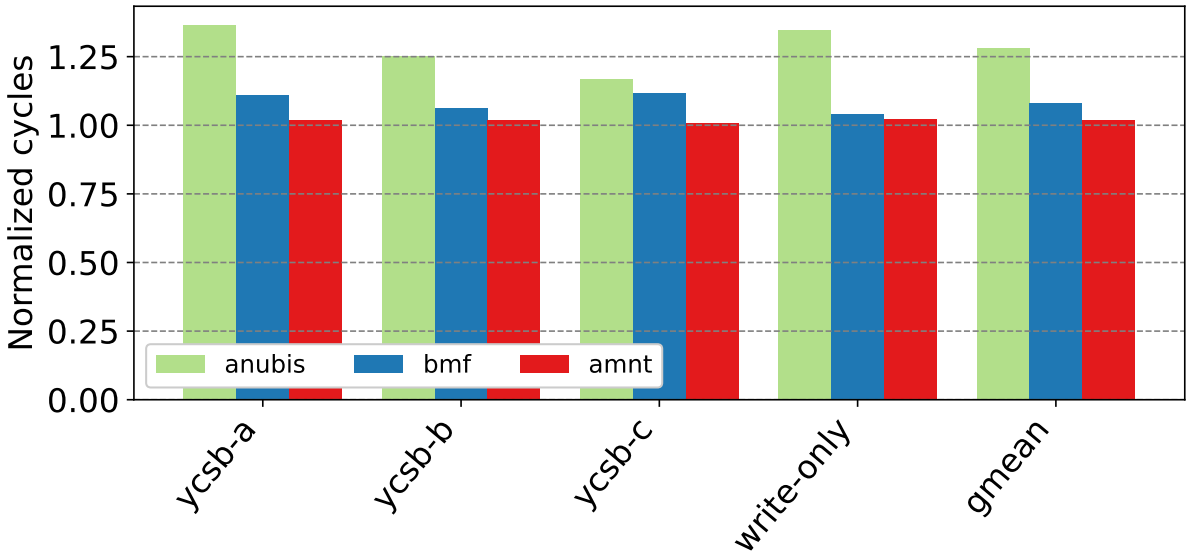


Figure 4.8: Cycles for the YCSB workloads in memcached server/client normalized to volatile secure memory. client has to perform small computation to determine the next key (and value on a write), neither of which are memory-intensive operations. However, AMNT still performs well relative to the baseline integrity-protection protocols. AMNT improves overhead by 24% compared to Anubis and by 2% compared to BMF.

4.7.4 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, we assume that the recomputation time is bound by the memory bandwidth. A single Optane DIMM supports around 4 GB/s of total bandwidth when subjected to a mixed read/write sequential workload [48], of which around half of this bandwidth (2 GB/s) is dedicated to reads. Assuming a six-channel machine [50, 43], this provides a total read bandwidth, at recovery, of 12 GB/s to memory, which is the essential performance bottleneck for recovery. We use this bandwidth to generate the data in Table 4.10a.

Table 4.10a shows the time it takes to recover each of the baseline and state-of-the-art configurations after a system failure. Note that strict and BMF protocols would have essentially zero recovery time so they are not shown in the table. In general, relaxing the metadata persistence model increases the percentage of nodes that will be stale on a system failure increasing recovery time.

Unlike prior approaches, the 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 [106] (as shown in Fig. 4.10a). Such recovery times are still much faster than other state-of-the-art alternatives, and the relaxation of metadata persistence implies that AMNT can outperform Anubis at runtime. In the event that a service provider cannot tolerate downtimes

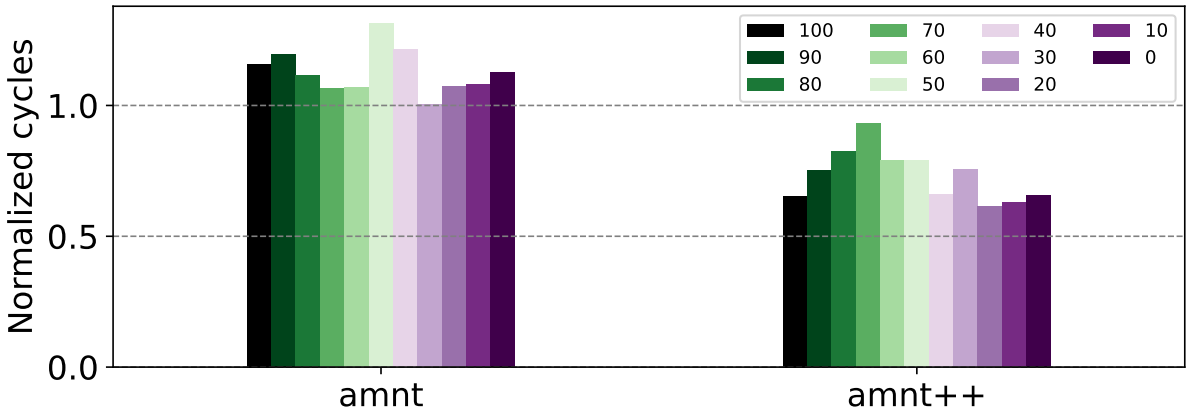
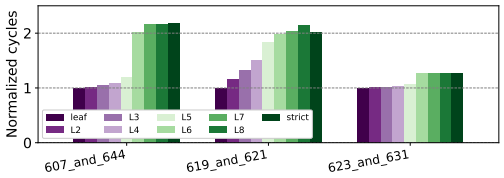


Figure 4.9: Performance for YCSB A for varied threshold values.

	256GB	2TB	16TB	128TB	BMT stale %
leaf	0.78 sec	6.22 sec	49.77 sec	6.64 min	100%
Anubis	1.3 ms	1.3 ms	1.3 ms	1.3 ms	fixed
Osiris	6.63 sec	50.6 sec	6.75 min	54.1 min	100%*
AMNT L2	0.09 sec	0.78 sec	6.22 sec	49.77 sec	12.5%
AMNT L3	0.01 sec	0.09 sec	0.78 sec	6.22 sec	1.56%
AMNT L4	0.002 sec	0.01 sec	0.09 sec	0.78 sec	0.2%
AMNT L5	.25 ms	0.002 sec	0.01 sec	0.09 sec	0.2%

(a) Recovery times for the different protocols as a function of memory size.



(b) Performance of AMNT by subtree level.

Figure 4.10: Sensitivity of AMNT by subtree level for recovery and performance under multiprogram workloads against [106, 103].

this long, 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 4.10a).

Fig. 4.10b demonstrates the overhead of the subtree at varying levels for each of the SPEC 2017 CPU benchmarks and multiprogram benchmarks, each of which has a different implication on recovery time. Placing the "subtree" at the true root of the tree implies that the entire BMT implements the leaf metadata persistence model, whereas L8 implies that only a single leaf node is protected by a subtree. The rest of the BMT implements strict metadata persistence. By default, we chose to put the subtree root at level 3 (i.e., 128MB will be stale at recovery) as it is the closest subtree to the leaves before performance gets scalably worse. This is a further demonstration as to why maintaining a relatively stable subtree root is beneficial to the overall performance.

4.7.5 AMNT++

We implement the kernel modifications in the Linux kernel v4.14. Our changes are implemented via a small patch to the kernel, which is composed of fewer than 150 lines of code in the memory management (`mm`) package.

In this evaluation, we prefill memcached with the whole key space to ensure that performance is not hamstrung by expensive allocations. Then, we measure 50,000 operations (reads and stores) according to YCSB workload A with uniformly random keys. We run AMNT with the subtree at level 5, which protects 2MB of contiguous memory. Given this configuration, most memory accesses will be a consequence of accessing the key-value store in the memcached server. Ideally the key-value store is physically contiguous

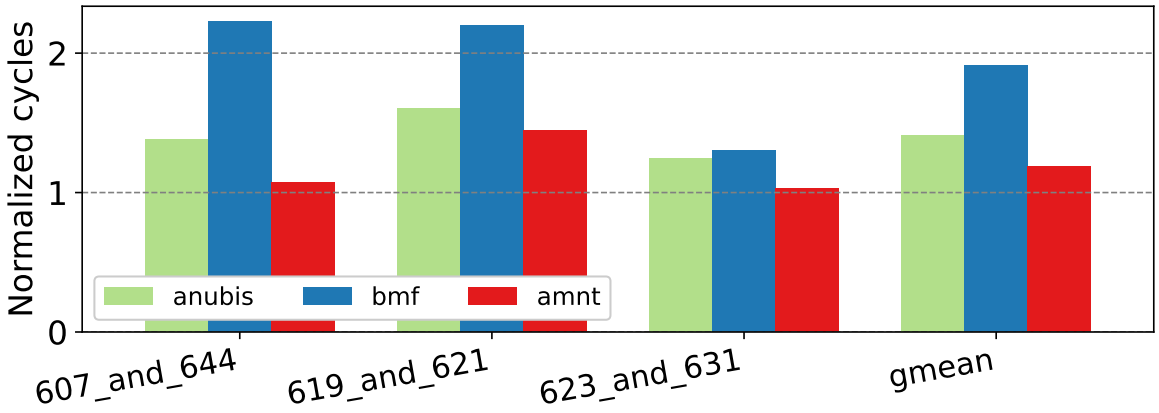


Figure 4.11: Normalized cycles for multiprogram workloads.

within a subtree.

We vary the hardware-defined threshold requirements to move the subtree in order to stress the impact of the subtree hit rates on performance. A 100% threshold implies that each of the n most recent writes to memory must fall within the same subtree to move subtree root tracked by AMNT to protect that subtree. This case implies a "sticky" subtree root, where movement is infrequent. A 0% threshold means that the subtree will always move to the most frequently accessed subtree after the n most recent writes to memory, without any regard for a minimum threshold.

Figure 4.9 shows the raw number of cycles executed. The modified OS demonstrates an approximately 30% reduction in running time versus the unmodified OS. We attributed the performance improvement between the modified and unmodified OS to increased physical spatial locality of the application. The modified OS increases subtree hit rates by about 20%. We attribute the rest of the performance improvement to shorter TLB walks from the increased physical locality, and the fact that fewer operations undergo the security protocol. AMNT++ results in a 70% reduction in LLC misses versus AMNT.

The re-organization of OS data structures can require a significant amount of time, and the physical page allocation structure is re-organized at a rate proportional to the physical page allocation rate. As such, in workloads that require significant physical allocations, such as booting the operating system, AMNT++ modifications result in significant overhead. For the Linux boot, an allocation and reclamation intensive procedure, we find that these OS modifications add an additional 54.7% runtime overhead. However, allocation intensive periods are predominantly at the inception of a process. For SCM workloads, which are typically long-running, this will not be the predominant bottleneck, and secure SCMs may benefit from the modified physical allocator.

4.7.6 Multiprogram Analysis

To further evaluate AMNT's ability to track hot regions in physical memory, we perform a multiprogram analysis. To do so, we select multiple SPEC CPU 2017 benchmarks and run them together. We first determined which benchmarks contain the most temporally similar regions of interest, and run them each on their own partition of CPUs. From our analysis, we find the following pairs need to be run together: *607.cactuBSSN_s* with *644.nab_s*, 631.deepsjeng_s with *623.xalancbmk_s*, and *619.lbm_s* with *621.wrf_s*. We pin one benchmark to 2 cores and the other benchmark to another set of 2 cores to isolate the runtime

overhead and remove the cost of context switches.

This setup has two key properties that will stress the ability of AMNT to track hot regions of memory. For one, two different running processes means that two different address spaces will be allocated. This behavior decreases the likelihood of physical contiguity and stresses the hotness tracking ability of AMNT. Furthermore, running multiple programs in parallel creates cache dissonance in the shared LLC. Activity from one benchmark may impact the cache behavior of the other. Such a scenario creates memory behavior that precludes either address space from being hotter than the other in physical memory.

AMNT outperforms state-of-the-art Anubis [106] in this setting by 20% and BMF by 40% (shown in Figure 4.11). Similar to the single program evaluation, we find that running multiprogram workloads do not impact subtree hit rate. The increased LLC dissonance negatively impacts the metadata cache locality more severely than it does the subtree hit locality.

4.7.7 Hardware Overheads

One of AMNT's goals is to limit the additional hardware components both on-chip and in memory. On-chip area is in high demand for various hardware optimizations across a multitude of workloads, so spatial overheads should be minimized. Furthermore, trends in secure memory have moved towards reducing the in-memory spatial overhead of secure memory, so increasing space in memory should be avoided as well. The hardware area overheads for Anubis, BMF and AMNT are listed in Table 4.2. In BMF, the non-volatile on-chip space consists of the non-volatile metadata cache. The volatile on-chip space is solely the metadata cache (64kB), but note that each cache line is modified with extra bits for frequency counters. In Anubis, the non-volatile on-chip space is occupied by the additional root required to track the shadow Merkle Tree. The volatile on-chip space is composed of both the metadata cache and, optionally, the shadow MT cache (37kB). AMNT has one non-volatile register on-chip to track the location of the fast subtree. Its volatile on-chip space is composed of a 768 bit history buffer, and a 128 bit dirty path bitmap, plus the metadata cache. As such, the 37kB shadow cache in Anubis is $49X$ bigger in capacity than the history buffer and dirty path bitmap in AMNT. The 4kB non-volatile metadata cache in BMF is $32X$ bigger than the additional subtree root in AMNT.

4.8 Related Work

4.8.1 Secure Memory

Secure memory is an active area of research, with significant improvements to the original secure memory design over the last two decades. The state-of-the-art secure memory solutions combine one-time-pads (OTP) and counter-mode encryption done in hardware [99, 36, 79, 8, 91]. For integrity verification, state-of-the-art include the use of modified Bonsai Merkle Trees (BMTs) for performance, fault tolerance and area

	NV On-Chip	Vol. On-Chip	In-Memory
Anubis	64 B	101 kB	37 kB
BMF	4 kB	65 kB	-
AMNT	64 B	65 kB	-

Table 4.2: Hardware overheads of the state-of-the-art.

optimizations. In Morphable Counters, Saileshwar *et.al.* [80] look to further reduce the area overhead of the integrity tree by introducing novel mechanisms for having a multi-arity tree that can be adapted based on the security and performance needs of the system. Synergy [81] is a mechanism that combines properties of secure memory along with Error Correcting Codes (ECC) in memory to compact the amount of memory overhead required to store the data HMACS. Most of these techniques are orthogonal to the proposed work and can be implemented in combination to the AMNT mechanism.

4.8.2 Hardware for SCM

Hardware for storage class memory, and in particular, for enforcing crash-consistency of data, has had two major themes in the literature: memory persistency and persistent transactions. The earlier theme focused on minimal hardware primitives for enforcing ordering of data updates into persistent memory [23, 54, 71, 52], broadly termed memory persistency [74]. This work was primarily oriented on models to describe the ordering and timing of writes-back from the volatile caches into the persistent SCM, in an analog to memory consistency models [2]. The later theme explored the support of transactional updates to persistent memory within the architecture [105, 51, 88, 16, 97], in the style of hardware transactional memory [42]. This work trades hardware complexity and generality for ease-of-use by the programmer and overall runtime performance, and can be seen as a supplementary architecture to the less expressive memory persistency work. Battery-backed caches [82] provide new challenges and opportunities for application persistence [104, 104, 102], and the persistence of security metadata [32, 46, 4], but knowing how much battery is required for data-dependent flushing remains an open issue. While these systems are, for the most part, generic infrastructure for providing crash-consistency at the application level, they inform the design of our system.

4.8.3 Secure SCM

Anubis [106], much like AMNT, provides low run-time overhead and fast recovery. Anubis tracks the precise address of each update to counters and Merkle Tree nodes through an additional shadow cache and designated section of non-volatile memory. Anubis maps the addresses of all metadata blocks resident in the volatile metadata cache to a specific region in the non-volatile memory designated as the "shadow table". This shadow table is used to know which nodes need to be recomputed in the case of a crash. The shadow table is also protected by a "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. 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.

Bonsai Merkle Forest (BMF) [32], the most similar of the prior work to AMNT, is a protocol designed to dynamically reduce the leaf-to-root write path for frequently accessed nodes. 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 metadata persistence. Unlike BMF,

AMNT does not assume full leaf coverage, so it can benefit in 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, which require more area volatile memories (i.e., SRAM caches) per byte stored, to cache the large number of roots required for full leaf coverage.

Osiris's [103] method of persistence hinges on reducing memory accesses when updating the tree in favor of cache updates, only persisting integrity tree leaves to memory every n updates, where n is a set interval. Persist Level Parallelism [33] 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 two works do not consider recovery time, and are unlikely to be implemented in real-world systems.

4.9 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. In this paper, we present *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. Emerging architectures beyond just non-volatile SCMs will continue to challenge the semantics of secure memory protocols.

Part III

Trust in Secure Memory

Chapter 5

Huffman

Chapter 6

Distributed Secure Memory

Bibliography

[1] Martin Abadi et al. "Control-flow integrity principles, implementations, and applications". In: *ACM Transactions on Information and System Security (TISSEC)* 13.1 (2009), pp. 1–40.

[2] S.V. Adve and K. Gharachorloo. "Shared memory consistency models: a tutorial". In: *Computer* 29.12 (1996), pp. 66–76. DOI: 10.1109/2.546611.

[3] Mohammad Alshboul, James Tuck, and Yan Solihin. "Lazy persistency: A high-performing and write-efficient software persistency technique". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2018, pp. 439–451.

[4] 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.

[5] 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).

[6] 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.

[7] Ittai Anati et al. "Innovative technology for CPU based attestation and sealing". In: *Proc. International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. 2013.

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

[9] Joy Arulraj, Andrew Pavlo, and Subramanya R Dulloor. "Let's talk about storage & recovery methods for non-volatile memory database systems". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 2015, pp. 707–722.

[10] Tuomas Aura. "Strategies against replay attacks". In: *Proceedings 10th Computer Security Foundations Workshop*. IEEE. 1997, pp. 59–68.

[11] 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.

[12] 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.

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

[14] 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.

[15] Nathan Burow et al. "Control-flow integrity: Precision, security, and performance". In: *ACM Computing Surveys (CSUR)* 50.1 (2017), pp. 1–33.

[16] Miao Cai, Chance C. Coats, and Jian Huang. "HOOP: Efficient Hardware-Assisted Out-of-Place Update for Non-Volatile Memory". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020, pp. 584–596. DOI: 10.1109/ISCA45697.2020.00055.

[17] Zora Caklovic, Product Expert, Oliver Rebholz, et al. "Bringing persistent memory technology to sap hana: Opportunities and challenges". In: *Annual SNIA Persistent Memory Summit* (2017).

[18] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. "Atlas: Leveraging Locks for Non-volatile Memory Consistency". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applicat ions*. OOPSLA '14. Portland, Oregon, USA: ACM, 2014, pp. 433–452. ISBN: 978-1-4503-2585-1. DOI: 10.1145/2660193.2660224. URL: http://doi.acm.org/10.1145/2660193.2660224.

[19] Shuo Chen et al. "Non-control-data attacks are realistic threats." In: *USENIX security symposium*. Vol. 5. 2005, p. 146.

[20] 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.

[21] Siddhartha Chhabra and Yan Solihin. "i-NVMM: A secure non-volatile main memory system with incremental encryption". In: *2011 38th Annual international symposium on computer architecture (ISCA)*. IEEE. 2011, pp. 177–188.

[22] Joel Coburn et al. "NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories". In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '11. Newport Beach, California, USA: ACM, 2011, pp. 105–118. ISBN: 978-1-4503-0266-1. DOI: http://doi.acm.org/10.1145/1950365.1950380. URL: http://doi.acm.org/10.1145/1950365.1950380.

[23] Jeremy Condit et al. "Better I/O through Byte-Addressable, Persistent Memory". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 133–146. ISBN: 9781605587523. DOI: 10.1145/1629575.1629589. URL: https://doi.org/10.1145/1629575.1629589.

[24] Rob Coombs. "Securing the future of authentication with ARM TrustZone-based trusted execution environment and fast identity online (FIDO)". In: *ARM White paper* (2015).

[25] 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.

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

[27] Stephen Crane et al. “Readactor: Practical code randomization resilient to memory disclosure”. In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 763–780.

[28] Morris Dworkin, NATIONAL INST OF STANDARDS, and TECHNOLOGY GAITHERSBURG MD COMPUTER SECURITY DIV. “Recommendation for Block Cipher Modes of Operation. Methods and Techniques”. In: (2001).

[29] Erhu Feng et al. “Scalable Memory Protection in the PENGLAI Enclave.” In: *OSDI*. 2021, pp. 275–294.

[30] Brad Fitzpatrick. “Distributed caching with memcached”. In: *Linux journal* 2004.124 (2004), p. 5.

[31] Fabian Fleischer, Marcel Busch, and Phillip Kuhrt. “Memory corruption attacks within Android TEEs: a case study based on OP-TEE”. In: *Proceedings of the 15th International Conference on Availability, Reliability and Security*. 2020, pp. 1–9.

[32] 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.

[33] 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.

[34] Blaise Gassend et al. “Caches and hash trees for efficient memory integrity verification”. In: *Proc. International Symposium on High Performance Computer Architecture (HPCA)*. 2003.

[35] 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.

[36] Shay Gueron. “A Memory Encryption Engine Suitable for General Purpose Processors”. In: *Proc. International Association for Cryptologic Research (IACR)* (2016).

[37] 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.

[38] Xijing Han, James Tuck, and Amro Awad. “Horus: Persistent Security for Extended Persistence-Domain Memory Systems”. In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2022, pp. 1255–1269.

[39] 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.

[40] 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.

[41] Michael Henson and Stephen Taylor. “Memory encryption: A survey of existing techniques”. In: *ACM Computing Surveys (CSUR)* 46.4 (2014), pp. 1–26.

[42] Maurice Herlihy and J. Eliot B. Moss. "Transactional Memory: Architectural Support for Lock-Free Data Structures". In: *Proceedings of the 20th Annual International Symposium on Computer Architecture*. ISCA '93. San Diego, California, USA: Association for Computing Machinery, 1993, pp. 289–300. ISBN: 0818638109. DOI: 10.1145/165123.165164. URL: https://doi.org/10.1145/165123.165164.

[43] 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.

[44] 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.

[45] Jianming Huang and Yu Hua. "A write-friendly and fast-recovery scheme for security metadata in non-volatile memories". In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2021, pp. 359–370.

[46] Jianming Huang and Yu Hua. "Ensuring Data Confidentiality in eADR-Based NVM Systems". In: *IEEE Computer Architecture Letters* 21.2 (2022), pp. 153–156.

[47] Jianming Huang and Yu Hua. "Root Crash Consistency of SGX-style Integrity Trees in Secure Non-Volatile Memory Systems". In: *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2023, pp. 152–164.

[48] *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.

[49] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. "Failure-Atomic Persistent Memory Updates via JUSTDO Logging". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and O perating Systems*. ASPLOS '16. Atlanta, Georgia, USA: ACM, 2016, pp. 427–442. ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872410. URL: http://doi.acm.org/10.1145/2872362.2872410.

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

[51] Arpit Joshi et al. "ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging". In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2017, pp. 361–372. DOI: 10.1109/HPCA.2017.50.

[52] Arpit Joshi et al. "Efficient Persist Barriers for Multicores". In: *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. Waikiki, Hawaii: Association for Computing Machinery, 2015, pp. 660–671. ISBN: 9781450340342. DOI: 10.1145/2830772.2830805. URL: https://doi.org/10.1145/2830772.2830805.

[53] 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.

[54] Aasheesh Kolli et al. "Delegated persist ordering". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016, pp. 1–13. DOI: 10.1109/MICRO.2016.7783761.

[55] David E Kotecki. “A review of high dielectric materials for DRAM capacitors”. In: *Integrated Ferroelectrics* 16.1-4 (1997), pp. 1–19.

[56] Tamara Silbergleit Lehman, Andrew Douglas Hilton, and Benjamin C. Lee. “MAPS: Understanding Metadata Access Patterns in Secure Memory”. In: *Proc. International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2018.

[57] Tamara Silbergleit Lehman, Andrew Douglas Hilton, and Benjamin C. Lee. “PoisonIvy: Safe speculation for secure memory”. In: *Proc. International Symposium on Microarchitecture (MICRO)*. 2016.

[58] 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.

[59] David Lie, Chandramohan A Thekkath, and Mark Horowitz. “Implementing an untrusted operating system on trusted hardware”. In: *Proc. Symposium on Operating Systems Principles (SOSP)*. 2003.

[60] David Lie et al. “Architectural support for copy and tamper resistant software”. In: *SIGPLAN Notices* (2000).

[61] 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.

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

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

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

[65] 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.

[66] David A McGrew. “Counter mode security: Analysis and recommendations”. In: *Cisco Systems, November* 2.4 (2002).

[67] Frank McKeen et al. “Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave”. In: *Proceedings of the Hardware and Architectural Support for Security and Privacy*. 2016, pp. 1–9.

[68] Stephen McLaughlin et al. “The cybersecurity landscape in industrial control systems”. In: *Proceedings of the IEEE* 104.5 (2016), pp. 1039–1057.

[69] Ralph C Merkle. “Protocols for Public Key Cryptosystems.” In: *Proc. Symposium on Security and Privacy (SP)*. 1980.

[70] 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.

[71] Sanketh Nalli et al. "An Analysis of Persistent Memory Use with WHISPER". In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '17. Xi'an, China: ACM, 2017, pp. 135–148. ISBN: 978-1-4503-4465-4. DOI: 10.1145/3037697.3037730. URL: http://doi.acm.org/10.1145/3037697.3037730.

[72] Bernard Ngabonziza et al. "Trustzone explained: Architectural features and use cases". In: *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*. IEEE. 2016, pp. 445–451.

[73] Ismail Oukid et al. "Instant Recovery for Main Memory Databases." In: *CIDR*. 2015.

[74] 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.

[75] Marcus Pendleton et al. "A survey on systems security metrics". In: *ACM Computing Surveys (CSUR)* 49.4 (2016), pp. 1–35.

[76] Erez Perelman et al. "Using simpoint for accurate and efficient simulation". In: *ACM SIGMETRICS Performance Evaluation Review* 31.1 (2003), pp. 318–319.

[77] pmem.io. *Persistent Memory Development Kit*. http://pmem.io/pmdk. 2017.

[78] Brian Rogers et al. "Single-level integrity and confidentiality protection for distributed shared memory multiprocessors". In: *Proc. International Symposium on High Performance Computer Architecture (HPCA)*. 2008.

[79] Brian Rogers et al. "Using address independent seed encryption and Bonsai Merkle trees to make secure processors OS- and performance-friendly". In: *Proc. International Symposium on Microarchitecture (MICRO)*. 2007.

[80] Gururaj Saileshwar et al. "Morphable Counters: Enabling Compact Integrity Trees For Low-Overhead Secure Memories". In: *Proc. International Symposium on Microarchitecture (MICRO)*. 2018.

[81] Gururaj Saileshwar et al. "SYNERGY: Rethinking Secure-Memory Design for Error-Correcting Memories". In: *Proc. International Symposium on High Performance Computer Architecture (HPCA)*. 2018.

[82] Steve Scargall. *Programming persistent memory: A comprehensive guide for developers*. Springer Nature, 2020.

[83] Felix Schuster et al. "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications". In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 745–762.

[84] Daniele Sgandurra and Emil Lupu. "Evolution of attacks, threat models, and solutions for virtualized systems". In: *ACM Computing Surveys (CSUR)* 48.3 (2016), pp. 1–38.

[85] Rifat Shahriyar et al. "Taking off the gloves with reference counting Immix". In: *ACM SIGPLAN Notices* 48.10 (2013), pp. 93–110.

[86] Weidong Shi and Hsien-Hsin S Lee. "Authentication control point and its implications for secure processor design". In: *Proc. International Symposium on Microarchitecture (MICRO)*. 2006.

[87] Weidong Shi et al. “Architecture support for high speed protection of memory integrity and confidentiality in multiprocessor systems”. In: *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2004.

[88] Seunghee Shin et al. “Proteus: A Flexible and Fast Software Supported Hardware Logging Approach for NVM”. In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-50 ’17. Cambridge, Massachusetts: Association for Computing Machinery, 2017, pp. 178–190. ISBN: 9781450349529. DOI: 10.1145/3123939.3124539. URL: https://doi.org/10.1145/3123939.3124539.

[89] Ofir Shwartz and Yitzhak Birk. “Distributed memory integrity trees”. In: *IEEE Computer Architecture Letters* 17.2 (2018), pp. 159–162.

[90] 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.

[91] G Edward Suh et al. “AEGIS: Architecture for tamper-evident and tamper-resistant processing”. In: *Proc. International Conference on Supercomputing (ICS)*. 2003.

[92] G Edward Suh et al. “Efficient memory integrity verification and encryption for secure processors”. In: *Proc. International Symposium on Microarchitecture (MICRO)*. 2003.

[93] 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.

[94] 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.

[95] Haris Volos, Andres Jaan Tack, and Michael M. Swift. “Mnemosyne: Lightweight Persistent Memory”. In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: ACM, 2011, pp. 91–104. ISBN: 978-1-4503-0266-1. DOI: 10.1145/1950365.1950379. URL: http://doi.acm.org.ezp.lib.rochester.edu/10.1145/1950365.1950379.

[96] 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.

[97] Xueliang Wei et al. “MorLog: Morphable Hardware Logging for Atomic Persistence in Non-Volatile Main Memory”. In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020, pp. 610–623. DOI: 10.1109/ISCA45697.2020.00057.

[98] 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.

[99] Chenyu Yan et al. “Improving cost, performance, and security of memory encryption and authentication”. In: *Proc. International Symposium on Computer Architecture (ISCA)* (2006).

[100] Jun Yang, Lan Gao, and Youtao Zhang. "Improving memory encryption performance in secure processors". In: *IEEE Transactions on Computers* 54.5 (2005), pp. 630–640.

[101] Jun Yang, Youtao Zhang, and Lan Gao. "Fast secure processor for inhibiting software piracy and tampering". In: *Proc. International Symposium on Microarchitecture (MICRO)*. 2003.

[102] Chongnan Ye et al. "Enabling Atomic Durability for Persistent Memory with Transiently Persistent CPU Cache". In: *arXiv preprint arXiv:2210.17377* (2022).

[103] 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.

[104] Bowen Zhang et al. "NBTree: a Lock-free PM-friendly Persistent B+-Tree for eADR-enabled PM Systems". In: *Proceedings of the VLDB Endowment* 15.6 (2022), pp. 1187–1200.

[105] 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*. MICRO-46. Davis, California: ACM, 2013, pp. 421–432. ISBN: 978-1-4503-2638-4. DOI: 10.1145/2540708.2540744. URL: http://doi.acm.org/10.1145/2540708.2540744.

[106] 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.

[107] Pengfei Zuo, Yu Hua, and Yuan Xie. "Supermem: Enabling application-transparent secure persistent memory with low overheads". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 479–492.