Proposal: A Study of Performance and Trust in Secure Memory

Sam Thomas

Abstract

When software engineers and developers write a program, they trust that hardware will execute it faithfully. Setting some variable "x = 5" will mean that "x" will always be "5" until the program changes its state because, well, why wouldn't it? Unfortunately, hardware is not perfect. Physical properties of devices can be exploited to tamper with the correctness of a program, and has given rise to a host of new security vulnerabilities. Main memory devices such as DRAM are becoming increasingly dense; a consequence of which being that the state of stored data can be modified without accessing that address through physical attacks like Rowhammer. Furthermore, emerging technologies like non-volatile memories make rebooting the system non-viable solutions. This is one of several reasons motivating secure memory hardware that can guarantee the privacy and integrity of values in memory.

Performance is the biggest challenge to practical secure memory. Intel SGX is an example of a commodity hardware that had implemented lightweight secure memory in its original release in 2015, and was significantly scaled back in 2021 due to inhibiting performance. Furthermore, emerging technologies and use cases, like non-volatile memories and distributed memory, challenge volatile, single-node semantics and pose even greater performance challenges. As a result, state-of-the-art work in secure memory has proposed varying caching strategies and structures in the trusted domain to help accelerate these protocols.

We are at a tipping point in secure memory design. These caches are becoming increasingly essential to creating low-overhead secure memory techniques. However, bloated trusted compute bases in secure memory designs may lead to unforeseen future vulnerabilities if they become too embedded into the design space.

In this thesis proposal, I will describe a series of secure memory protocols and optimizations targeted at emerging technologies and modern use cases. Unlike prior art, however, I treat trust as a first-class design principle much like performance. In particular, I will describe: (1) A Midsummer Night's Tree, a secure memory protocol for non-volatile memories that reduces performance overhead by 41% and auxiliary trusted device capacity by $32-49X$ versus the state of the art; (2) a cache-free Huffmanized Merkle Tree, an alternative approach to secure memory design that fundamentally re-thinks the construction of secure memory to minimize trusted hardware without sacrificing performance; (3) a distributed caching policy for secure memory metadata in CXL systems, which leverages data-center workloads coupled with the fact that target devices for secure memory will likely exist in multi-node settings to extend the "effective" cache sizes without requiring extensive hardware.

I. Introduction

The way that we interact with compute resources has drastically changed in recent times. Compute is increasingly remote, and data resides on servers and in memory that end-users will rarely see at a similarly increasing rate. This means that providing security and privacy guarantees to the end-user is increasingly difficult. An unbounded number of legitimate alternative guests are running programs and storing data in parallel; servers likely coordinate telemetry and workload monitoring in the system; warehouse managers are likely running monitoring scripts with root privileges. Each of these are potential vectors for vulnerabilities. Security conscious software may run with certain software mitigations (i.e. stack canaries, ASLR, etc.) [1], each of which escalates the required capability of an attacker for them to trigger unwarranted faults in a program's execution.

However, a software defense is only as effective as the hardware that underlies it. The emergence of certain vulnerabilities of hardware components, particularly in main memory devices (i.e., Rowhammer [2], [3], [4], etc.), means that values in memory can be corrupted maliciously [5] and targeted to illegally escalate program privileges [6] or hijack the operating system [7], [8] without accessing them. Furthermore, main memory is attached to the motherboard externally through a bus rather than being fabricated on the processor die. This means that anyone with physical access to the device can detach the memory module from the system and re-attach it elsewhere to read the contents of memory. Given this, privacy and integrity guarantees from the hardware are critical for a secure memory system.

Secure memory can be described by a protocol that dates back to the late 1990s and early 2000s [9], [10], [11], [12], [13], [14], [15], [16]. In particular, the protocol encrypts values to memory using temporarily and spatially unique counters in a protocol called *counter-mode encryption*. These encryption counters are used as input to an on-chip AES engine to produce a one-time pad, and this is XOR-ed with the plain-text data en route to it being stored in memory. Thus, data never resides in memory in plain-text to ensure its confidentiality. To provide integrity, a tree of hashes built on top of the plain-text data is stored in memory alongside the cipher-text data and the encryption counters. The root of this tree is stored on the processor chip to ensure that it hasn't been tampered with. Then, to fetch some data from memory, its cipher-text is fetched with the encryption counter. The one-time pad is reproduced and XOR-ed with the cipher-text to reproduce the plain-text. In parallel, the hash of the data and its ancestry through the integrity tree (i.e., its leaf-to-root path) is fetched. The hashes can then be computed from the data and compared against the stored hashes. If the hashes verify, then the hardware can conclude that the data hasn't been tampered with.

Seeing as this protocol now requires multiple fetches per data fetch, it comes with pretty significant overhead. Furthermore, the computation of hashes are dependent on the prior result. As such, fetching values from secure memory can become a significant bottleneck for memory intensive applications. Given this, an overwhelming majority of secure memory protocols in the literature take advantage of a cache reserved for the secure memory metadata [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32]. This cache is beneficial for two reasons: for one, cache hits provide faster access to values than going to main memory; furthermore, the cache resides on the processor chip, so hitting a value in the metadata cache means that value can behave as a root of trust, and authentication can stop in advance of reaching the integrity tree root.

Emerging technologies, like non-volatile main memories, distributed memory, software-defined far memory, etc., challenge existing secure memory semantics. These technologies have properties that either challenge the correctness of the secure memory protocol, provide opportunities for optimization, or both. For example, non-volatile main memory implies that care is required to ensure consistency between values in the volatile metadata cache on-chip and potentially stale values in memory in the event of a crash. Distributed memory raises questions about which host in a distributed shared memory system is responsible for providing secure memory semantics, whether or not secure memory metadata needs to be coherent, and how to make each of these work correctly. Software-defined far memory [33] describes a means of compressing infrequently accessed plain-text to increase the effective size of memory, and its interaction with the secure memory protocol has potentially interesting implications on performance.

We are reaching a tipping point in secure memory design. These emerging architectures have led to a new age of secure memory protocols and optimizations. In recent art, an overwhelming majority of secure memory literature diagnoses performance as the key limitation of secure memory hardware [34], [35], [36], [37], [28]. It is no secret as to why – this recent boom in secure memory literature directly corresponds to the timeline of the production of commodity devices that implemented secure memory in the form of Intel SGX [38], but these protections were removed in later SGX iterations only a few years later due to limiting performance [39], [40]. In doing so, several works propose increasing the area of on-chip components in addition to the metadata cache designated towards secure memory optimization. Each new on-chip component increases the number of trusted devices in the secure memory protocol. While optimizing for performance is important in secure memory, it's also important to consider the potential implications of these design decisions on trust in the secure memory protocol.

In this dissertation proposal, I will describe my work in secure memory. In particular, my work targets the implications of varying **emerging architectures** on secure memory. My work considers secure memory optimizations from the view of data structures, coherence protocols, application behaviors, etc. to achieve **low performance overhead** while considering **trust as a first-order design feature**. My work considers varying degrees of radical changes to the fundamental description of secure memory, and in doing so my work achieves significant performance benefits and reductions in on-chip area overheads.

This document is organized as follows:

1) Sec. II provides a more in-depth description of the secure memory protocol, and provides a foundation for the description of my work in the area.
2) Sec. III describes AMNT, my work in optimizing the secure memory protocol for non-volatile main memory devices while reducing on-chip area overhead in the trusted compute base by leveraging subtrees of the integrity tree and application behavior.
3) Sec. V describes the Baobab Merkle Tree, my work in leveraging encryption counter behavior to reduce the in-memory overhead of the integrity tree by memoizing these counters.
4) Sec. IV describes the Huffmanized Merkle Tree, my work in finding an optimally efficient, application-aware secure memory protocol that has the minimal set of trusted hardware.
5) Sec. VI describes my plans for what I would like to achieve in advance of my thesis defense.

I believe that there are several interesting problems that I would like to begin to pursue. Broadly, this describes work that I would like to pursue with respect to distributed secure memory given the emergence of CXL (Compute Express Link) [41]. In particular, I believe that this area is still very green, and ideas across works typically lack consistent assumptions and threads of thought. These works focus on the plausibility of secure memory in a distributed environment under conditions like RDMA or unified and coherent distributed address spaces. As such, I believe there is significant room to clarify and formalize this space. Furthermore, I hope to describe my intended contribution to this space in the form of a distributed secure memory metadata cache policy informed by my clarifications.

I am also looking forward to continuing my ongoing work with respect to AMNT, the Baobab Merkle Tree, and the Huffmanized Merkle Tree. AMNT and the Baobab Merkle Tree are currently under (re)submission, and I am targetting a fall submission for the Huffmanized Merkle Tree.

II. Background

In this section, 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.

A. Threat Model

Recent literature in the secure memory community generally assume a consistent threat model within a single memory device [30], [42], [43], [28], [37], [18], [44], [29], [22], [45], [24], [46], [47], [31], [48], [49], [50]. 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 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 [4], 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.

B. 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 [51], [52], [53], [54] (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 [54], [55]. 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

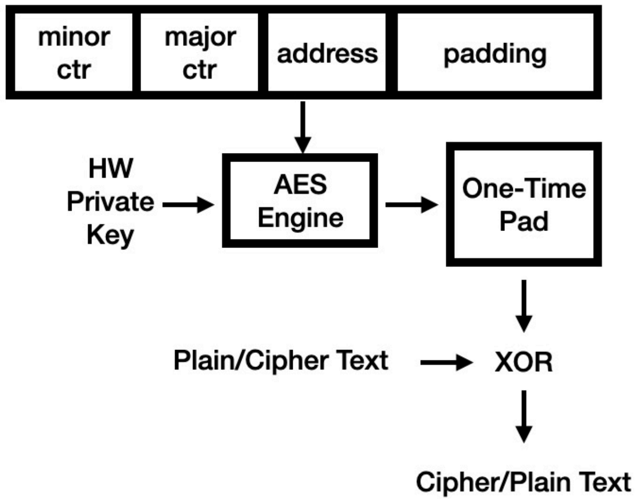


Fig. 1: Data-specific values are input to an AES engine to produce one-time pad. One-time pad is XORed with plain/cipher text to produce the inverse.

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 [35]. 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).

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 [56], [50], [57], [58], [59], [60], [61]; and (2) improving CME by modifying the layout of encryption counters [62], [49], [63], [64]. Furthermore, as memory architectures develop and new emerging technologies gain traction, the literature adapts to the new requirements held by these technologies [20], [27], [65], [66]. For example, non-volatile main memories have implications on how and when counters may be cached on-chip [27].

C. Integrity

Similarly, secure memory systems need a means of guaranteeing the integrity of data in memory. To do so, they typically implement an *integrity tree*. The motivation behind the integrity tree can be best described with a straw-man argument of how we can provide the integrity of a single 64-byte block of data at address a.

Suppose the memory hierarchy demands a fetch for a, and the value is fetched from memory. The hardware can store the *hash* of the data at a on-chip, in the trusted compute base. Storing hash h of the data at a ensures that the data in memory is not stored anywhere in plain-text, and compresses the storage overhead of preserving the state. After fetching a, compute the hash of the data and compare it to h. If they match, then the integrity of a is guaranteed.

Unfortunately, storing the hash of all data blocks in memory on-chip is not feasible due to spatial limitations. There is, however, space in main memory to store these blocks. As such, an alternative protocol could be one in which the hashes of each of the data blocks is stored in main memory rather than on-chip. This ensures that each block, rather than one, can have its integrity preserved. To authenticate some data, the hardware can fetch both a and the address of h from memory in parallel. Then, once both values are fetched, compute the hash of the data at a and compare it to h.

While this system works to a certain extent, there are certain limitations that require explicit addressing. For one, it is important that these hashes are one-way keyed hashes that are computed with a hardware private key. If not, an attacker can send some known plain-text values to known locations in memory and examine the hash state. Without using a one-way keyed hash, the attacker can reverse engineer the hardware private key by creating several data points of plain-text and then read the memory contents of the associated hash function.

Furthermore, and of greater concern with respect to the guarantees made by the hardware, an attacker can perform replay attacks on data and metadata in memory. For example, suppose the attacker is monitoring a at some time t_0 and sees that it is in state s_0 with associated hash h_0. Then, at some later t_1, the value at a is updated to s_1 and is now protected by h_1. If the attacker is capable of corrupting both the data at a and its associated hash, then the attacker can replace a with s_0 and h with h_0. The hardware, which only checks that some data can be verified by its hash, would see that the hashes verify and pass the corrupted values to the processor-side. The source of this vulnerability is that, seeing as all of the authentication metadata resides outside of the trusted hardware, there is nothing from which the hardware can build a root of trust. This is not an issue in the previous example because the hash was stored on-chip and, due to the threat model, could not be tampered.

In order to defend against the vulnerability of replay attacks in memory, the hardware needs to store some metadata on-chip so as to have some means of authenticating. This metadata needs to be small, so as not to occupy a significant portion of the limited on-chip area. Such is the motivation for the integrity tree. One of the earliest proposed integrity tree organizations is a tree of hashes such that an inner node of the tree is composed of the concatenated hashes of its children [9], [67]. The root of the tree, which is small (i.e., 64B), can be stored on-chip and act as a root of trust. Thus, verifying some data requires verifying its hash against its parent node and repeating this process until it is compared against the root. This series of hash computations reduces the likelihood of verification due to hash collision, as the likelihood of multiple successful collisions is extremely low.

This original design is functional in terms of providing integrity, but is not necessarily suitable for memory architecture for a few reasons. For one, this tree layout occupies a lot of space. For 8GB of main memory, there must be an 8-byte hash for every word (64B) of data in the leaf level alone. Further, to ensure that counters in memory haven't been corrupted, there also needs to be a subtree of this tree that protects the integrity of encryption counters. In total, for an 8GB memory, it requires 1.145GB of integrity tree nodes to protect tree nodes, which is an area overhead of 14.3%.

Beyond just area, however, is the fact that there are heavy performance limitations to this particular design. Consider the authentication protocol that was described before. We must authenticate some data by decrypting it, and comparing its hash against the stored hash to ensure that it hasn't been corrupted in memory. However, comparing against the hash now requires several accesses, as we ultimately need to compare against a trusted hash (i.e., the integrity tree root). We must *also* authenticate the counter value against the tree. If the tree is n levels tall, this means that we have $2n + 1$ times more memory accesses per data access in the secure memory case than in the traditional memory case.

To address this increased bandwidth requirement of memory to handle secure memory, one of the first optimizations to the secure memory protocol was the inclusion of a secure memory metadata specific cache [17]. This cache has two primary benefits. For one, cache hits have lower latency than memory accesses and require less bandwidth in memory due to the traffic in the cache. That is, parallel misses for the same address can be handled by caching protocols like MSHR, and cache hits don't require a memory fetch for that address. Furthermore, because this cache resides on-chip, values that hit in the cache are trusted. As such, authentications do not need to be computed up to the root, but instead only up to the first cache hit. This significantly reduces the number of metadata fetches required on the whole. Metadata specific caches have grown to be increasingly common in proposed secure memory designs [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], and several works have even proposed metadata type-specific caches.

Beyond the metadata cache, one of the most influential works in the secure memory literature was the proposal of the Bonsai Merkle Tree (BMT) [35]. The BMT, depicted in Fig. 2, addresses both the issue of performance *and* of spatial overhead. A BMT is similarly a tree of hashes, but the leaves of the BMT are *only* the encryption counters. That is, data is *not* protected by the BMT. Instead, data is protected by its MAC (keyed hash function, with the data and encryption counter as input) alone. To authenticate some data using the BMT, the cipher-text data is fetched from memory, and is decrypted using the untrusted encryption counter. The encryption counter has its integrity verified against the integrity tree (in the same way as in the tree in [68]). This implies that the counter value is trusted. Then, the MAC of the plain-text data is fetched, and the hash of the decrypted plain-text is computed.

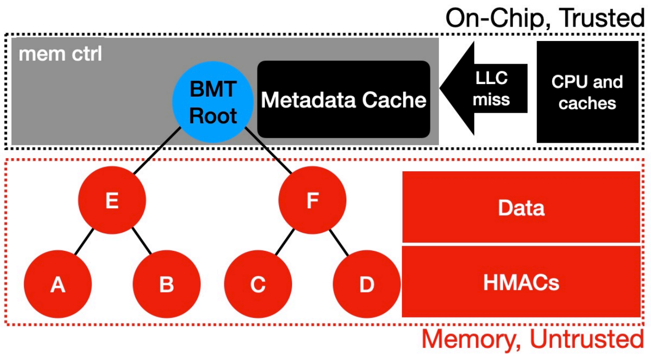


Fig. 2: A Bonsai Merkle Tree. Nodes are the hashes of their children, and leaves are encryption counters.

As proved in [35], if these hashes verify one another, then it must be the case that none of the cipher-text data, the encryption counter, nor the MAC of the plain-text has been tampered. To sketch this proof by contradiction, consider how an attacker would be able to modify some data such that some tampered value is interpreted as untampered by the secure memory hardware. It must be the case that the attacker has replayed some old values such that each of the encryption counter, cipher-text data, and plain-text data MAC verify one another. Can this be the encryption counter? If the attacker can use an old encryption counter, then they can replay the cipher-text data and plain-text data MAC to reset each of these values. However, the encryption counters are protected by the integrity tree, which is backed by a trusted value on-chip. As such, this value cannot be replayed. Can the attacker tamper the cipher-text and MAC? If they do so, then legitimate encryption counter will decrypt the tampered cipher-text into non-sense. There is a small chance that there will be a hash collision with this non-sense value and the plain-text data MAC, but this chance is very small and random. Thus, such a targeted attack would be difficult to carry out. Furthermore, replaying old MACs is unfeasible as the unique encryption counters produce different MAC values on each update, even if the plain-text value is the same.

A consequence of this structure is that the memory access requirements to authenticate some value in secure memory are reduced significantly. Now, only counters need to be authenticated against the integrity tree. This means that the number of memory accesses per data access in an n level tree is $n+2$, which is significantly less than in the integrity tree from [68]. This approach can still benefit from the metadata cache to further improve performance. Furthermore, the fact that the BMT is built on top of encryption counters alone significantly reduces the in-memory spatial requirements of the integrity tree. This is because the spatial granularity of encryption counters is 64B for every 4kB of data, whereas the spatial granularity of leaves in the tree in [68] is 64B for every 512B of data. That is, the BMT is 8X more compact than the prior model. Now, the spatial overhead of the BMT primarily comes from plain-text data MACs, which are still at the granularity of 64B for every 512B of data.

The other prominent integrity tree design in the literature today is the Parallelizable Integrity Tree (sometimes referred to as the SGX-style tree) [69], [11]. As the name implies, this is the integrity tree that was deployed in the original release of Intel SGX [38]. In this tree, nodes store a series of eight *nonces* (i.e., counter values) and a single hash, which is depicted in Fig. 3. Nonces are 7-byte values and refer to the number of updates in a node's child. The hash that is stored in the node is a keyed hash of the current state of the nonces, and uses its associated nonce in its parent as input to the hashing engine. That is, if a node is the i-th index child of the parent, then it will fetch the parent nonce at index i and use that value as input to the hashing engine to produce the hash in the node.

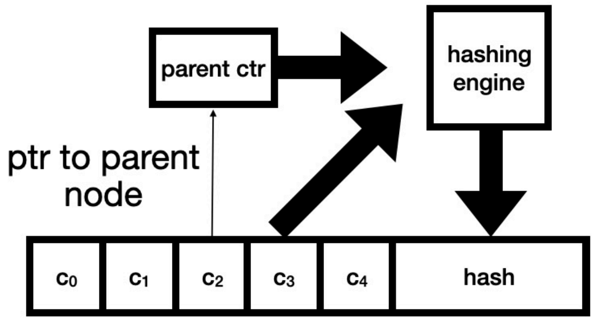


Fig. 3: SGX-style integrity tree node. Nodes store their hash, which takes the parent counter as input. As such, updating the tree only needs to occur up to a cache hit.

There are a few features of this integrity tree organization that are advantageous. For one, writing a value to memory no longer needs to be propagated up to the root of the integrity tree. Instead, on updating a tree node only requires writing up to the first cache hit. This is because the node in the cache is inherently trusted, so its hash isn't required to authenticate it in a greedy manner. Instead, on eviction, the block has its hash updated in advance of writing it back to memory, which results in a lazy protocol. Furthermore, this design allows for multiple different children to be updated in parallel, and only requires a single hash to be computed. That is, if m children are updated in parallel, then the nonce in the parent node will incremented by m. However, there only needs to be a single entry to the hashing engine per node. On hash computation, the hashing engine searches the input buffer for a more recent parent nonce before computing the hash, and if it exists it uses that value. This lazy design exhibits features from older work [69], and was suitable enough to be put in commodity hardware devices [38].

Despite the advances in reducing the performance overhead of secure memory, it still remains a fundamental bottleneck. In later iterations of SGX, secure memory guarantees were significantly reduced in favor of optimizing for performance [40]. Furthermore, new architectures and emerging memory technologies have provided the ground work for a new wave in the secure memory literature. There have been several works targeting crash recoverability of secure non-volatile main memories [19], [20], [21], [22], [18], [23], [24], [27], [29], [30], [32], [70], near-memory processing as a form of accelerating the secure memory protocol [71], [72], distributed secure memory [73], [74], [75], [42], [76], etc.

III. AMNT

As described in Sec. I, 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 [77]. 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 [78], [79], [80], [52]. 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 [81], [82], [83], escalate process privileges [84], [85], or any number of other memory corruption-based attacks [86], [87], [88], [89]. 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 [90], [91], [92].

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.

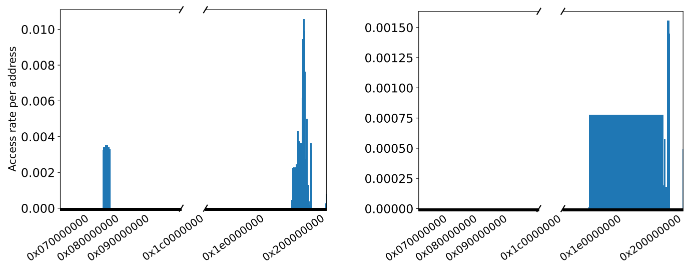
An alternative approach, dubbed *leaf metadata persistence* [27], addresses the performance issue by taking a *lazy* approach to crash consistency. That is, only p_t, H_t, and C_t are persisted directly at runtime. The tree nodes T_t are written to the volatile metadata cache and only written back to 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) [45]. For example, several works assume large metadata caching structures [20], [18], [93], [24], [46], [94] or assume physically large, NVM-based devices on-chip [22].

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 [31].

In this section, 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, 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.



(a) Trace of *perlbench*. (b) Trace of *lbm*.

Fig. 4: Memory accesses per address in different SPEC CPU 2017 benchmarks.

In this work, I make the following contributions:

- I present AMNT, a novel and efficient mechanism to persist security metadata alongside data.
- I 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.
- I describe a low-overhead recovery mechanism for SCM.
- I 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.
- I demonstrate how a system administrator can bound the recovery time using our proposed approach to achieve desired performance goals.

A. 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 [95] 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 [96], 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. III-E.

In Fig. 4, we show memory traces of two workloads that exhibit different characteristics. *Perlbench* (Fig. 4a) 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. 4b) 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

[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."

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.

B. Design

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 [80]. As discussed in Sec. III-A, 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.

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. 5). The rest of the BMT implements a strict metadata persistence strategy to minimize recovery time after a crash (red nodes in Fig. 5). 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.

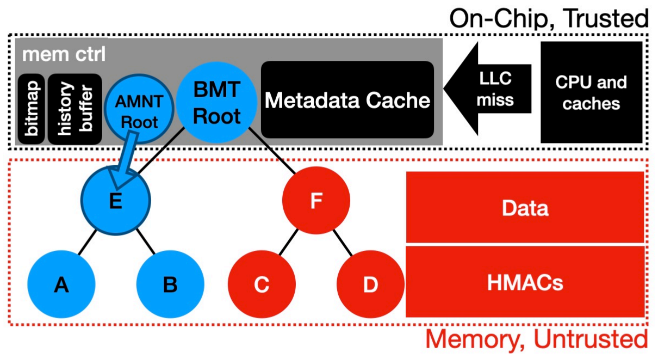


Fig. 5: A Midsummer Night's Tree. Red nodes implement strict persistence. Blue nodes implement leaf persistence.

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 [29]. 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.

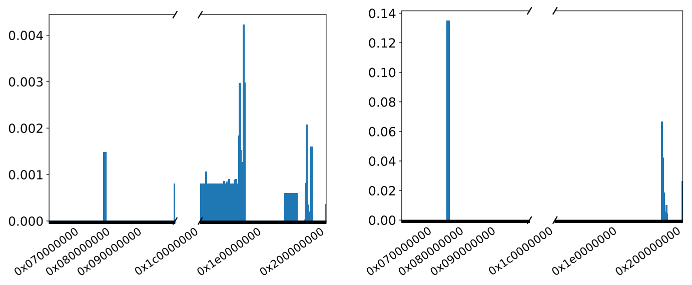
The history buffer (and corresponding bitmap) is an efficient and lightweight method to track the most frequently used regions of memory to select the best subtree root. Each entry in the history buffer requires at most $log_2 n$ bits for the region's index and an additional $log_2 n$ bits for the counter, resulting in $n * (2log_2 n)$ 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.

C. *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 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.

D. *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.



(a) Trace of multiprogram. (b) Multiprogram, modified OS.

Fig. 6: Memory accesses per address in *cactuBSSN* and *nab* for unmodified and modified physical page allocators.

In order to further increase in-memory physical locality, we modify the buddy allocation from the Linux operating system [97]. 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 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).

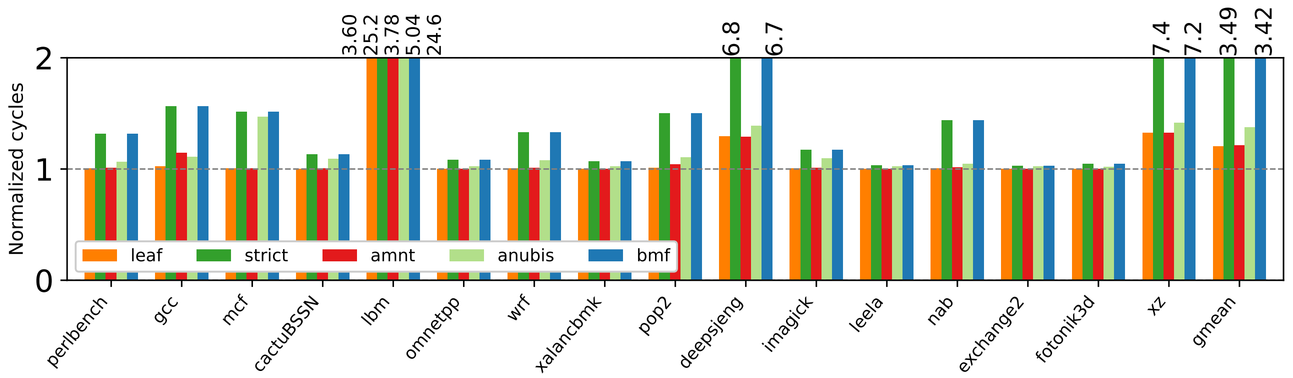


Fig. 8: Runtime comparison of AMNT, Anubis, and BMF protocols for the SPEC 2017 CPU benchmarks normalized to writeback secure memory protocol. Lower is better.

1) 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 shows idealistic scenario. To create more noise in the system, we ran multiprogram workloads (described in more detail in Sec. III-E), and show them in Fig. 6a.

Our modified OS increases the physical contiguity of hot regions of memory. Fig. 6b 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.

Fig. 7: 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 [98], 391ns write [99]

E. Evaluation

1) Methodology: We implement AMNT as an extension to gem5 [96], 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 [98], [99]. This configuration is similar to the one assumed in prior work on secure memory SCM [20], [30], [27]. We assume a specialized 64kB metadata cache for the security metadata as exists in Intel SGX [45]. The full configuration details are described in Table 7.

We evaluate AMNT against several state-of-the-art approaches in the BMT as implemented in [80] as well as two persistent protocols (leaf and strict) that represent the two extremes between runtime performance and recovery time. Our tests include:

1) **AMNT** our hardware implementation described in Sec. III-B. All evaluation referring to "AMNT" refers this implementation.
2) **AMNT++** our AMNT design including the modified OS described in Sec. III-C.
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** [20]: 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) [22]: 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 [20] assumes that the shadow table is tracked in a separate volatile on-chip cache to reduce the number of memory writes. Similarly, BMF [22] 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 [45].

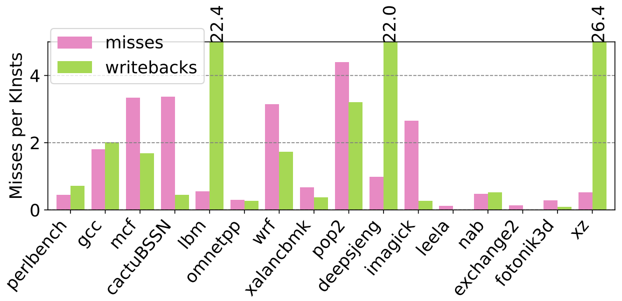


Fig. 9: Misses in LLC per 1000 instruction for the SPEC CPU 2017 benchmarks.

2) The SPEC CPU 2017 Suite: We evaluate AMNT on the SPEC CPU 2017 benchmark suite [95]. We run the speed benchmarks with ref inputs, and we fast-forward to a region of interest as determined by SimPoint [100] in the benchmark before simulating 500 million instructions to sample the execution. This approach is consistent with prior work [20], [27], [30], [29], [37].

Figure 8 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 [20]. 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 8).

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 9 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.

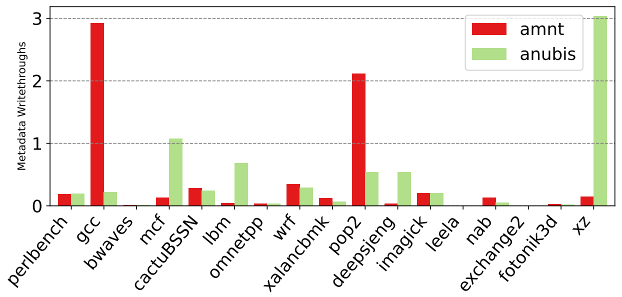
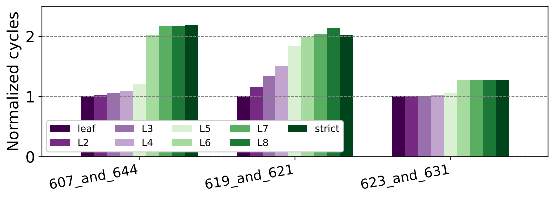


Fig. 10: Number of metadata writethroughs for AMNT and Anubis per 1000 instructions.

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 10). 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

	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.

Fig. 13: Sensitivity of AMNT by subtree level for recovery and performance under multiprogram workloads against [20], [27].

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).

3) Memcached with YCSB: We use memcached [65], 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 [101] 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 11 are less significant. This is largely due to the fact that the memcached server performs significant statistical logging, and the memcached 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.

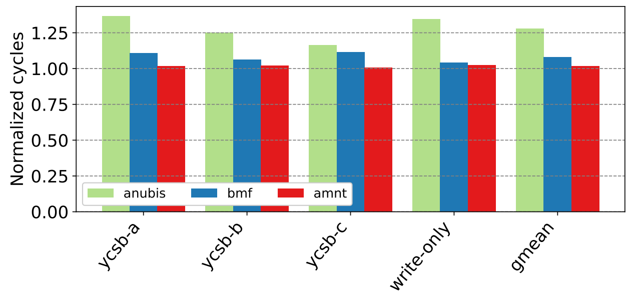


Fig. 11: Cycles for the YCSB workloads in memcached server/client normalized to volatile secure memory.

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 [102], of which around half of this bandwidth (2 GB/s) is dedicated to reads. Assuming a six-channel machine [98], [99], 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 13a.

Table 13a 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

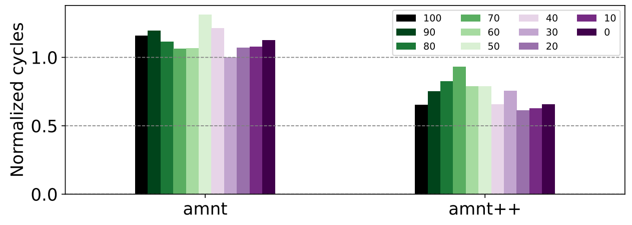


Fig. 12: Performance for YCSB A for varied threshold values.

configured at level 3 of the BMT, it has a slower recovery time than Anubis [20] (as shown in Fig. 13a). 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 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 13a).

Fig. 13b 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.

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 within a subtree.

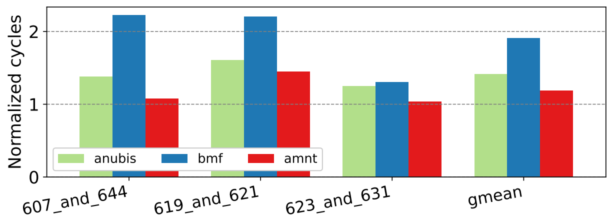


Fig. 14: Normalized cycles for multiprogram workloads.

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 12 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.

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.

	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	-

Fig. 15: Hardware overheads of the state-of-the-art.

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 [20] in this setting by 20% and BMF by 40% (shown in Figure 14). 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.

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 15. 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.

F. Summary

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 section, I presented *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.

IV. Huffmanized Merkle Tree

As modern computation workloads develop and change, we are finding that they are increasingly demanding on memory devices. To adjust for these workloads, DRAM devices and non-volatile memories have become increasingly dense. This makes them similarly increasingly vulnerable to physical exploitataions and attacks like Rowhammer [2], [3]. At the same time, purchasing these devices tend to be expensive, and a result there has been an increase in cloud computation and remote data storage. While renting time on remote machines allows for scalable computation for end-users, it also means that there is a largely unknown and potentially malicious set of other users on the same device that might exploit physical device vulnerabilities to malicious tamper with the execution of end-user programs.

All of this provides motivation for securing data stored in vulnerable off-chip memory devices. *Secure memory* describes a protocol in which data is encrypted in memory and the integrity of values in memory are protected via *integrity tree* (also *Merkle tree*) to ensure that they have not been corrupted in memory, and that no execution on malicious values can occur. That is, data values in memory are protected by a hash. These hashes, which also reside in memory and are subject to similar corruption attacks. As such, their integrity must also be protected, and so they are concatenated with physically close hashes, and these values together are again hashed to have their integrity protected. This process is repeated until a single hash remains, which is trusted because can be stored on-chip and reflects the overall state of memory. To authenticate a value in memory, its hash is computed and compared against the stored hash. This process is repeated recursively until it can be compared against a trusted value.

One of the biggest challenges of practical secure memory hardware is the impact that authenticating data values has on performance. In particular, to fetch a piece of data in memory, approximately ten additional metadata fields must be fetched, and have some computation performed on for decryption and integrity verification. To address the problem of performance, secure memory hardware is typically assisted by a small, specialized cache for optimizing the secure memory protocol [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32]. This cache is beneficial for multiple reasons – for one, it means accessing cached metadata values is fast, and going to main memory to fetch the value can be avoided. More importantly, however, the metadata cache resides on-chip, and it has been trusted due to its density (i.e., SRAM as opposed to DRAM). Commodity devices, like Intel SGX [103], have provided such secure memory protections in the past, and have been assisted by this security metadata cache [38]. Recent art has explored further optimizations to the secure memory protocol as, even with a metadata cache, secure memory protections are still execution bottlenecks for memory intensive applications.

However, there is a danger to the increasing reliance on the metadata cache towards optimizing the secure memory protocol. Consider the development of speculative execution in optimizing performance in processor design [104], and the increasing concern over the set of Spectre and Spectre-style attacks predicated on this optimization [105]. Speculative execution was such a performance advantage to processor design that it became fundamental to nearly all processor protocols for several decades. Yet, security implications of speculation were not considered as first-class issues in its design, and we now find ourselves in a situation where speculation is one of the most fundamental underlying vulnerabilities of modern processors. In this paper, we aim to avoid a world in which metadata caches become so fundamental to secure memory hardware that it is impossible to go back to a world before security metadata caches, similar to where we are today with speculation. To the best of our knowledge, there have not yet been any attacks predicated on exploiting the metadata cache in secure memory hardware. However, on-chip cache side-channel attacks like flush-and-reload [106] and prime-and-probe [107] demonstrate that they are not unreasonable to consider.

Metadata caches are effective for performance because they significantly reduce the amount of metadata required per data. That is, because values in the metadata cache are trusted, data authentications need to only be done up to a metadata cache hit. This proves to be fundamental to reducing the bottleneck imposed by secure memory, and is an important lesson in re-considering the role of the metadata cache. As such, an effective high-performance secure memory protocol should similarly reduce the number of metadata required to be accessed per data.

To achieve this end, we turn to the observation that not all data in memory is accessed at the same rate. This observation is not unique [22], [108], [109], [21], and it provides an opportunity to significantly reduce the amount metadata required to be accessed per data access – even without a metadata cache. In particular, we propose *huffmanizing* the Merkle Tree. Huffman trees are trees constructed with the frequency access related to the data in mind in order to achieve provably optimally shortest path lengths through the tree with respect to the frequency of access. As such, if path lengths through the Merkle Tree were provably optimally short with respect to the frequency with which some data is accessed, then we can achieve a similar end as providing performance secure memory semantics with a significantly reduced trusted compute basis.

Implementing the Huffmanized Merkle Tree presents several practical implementation issues. For one, the Huffmanized Merkle Tree is enforced by the hardware, so it has no knowledge of application semantics. Thus, it should be *dynamic* in nature to adjust its shape as the rate of access changes. This means that the frequency of access should be fast and computed on both reads and writes, whereas prior art in frequency-based secure memory optimizations has been previously based on writes only [22]. Relatedly, as the Huffmanized Merkle Tree updates its shape and state, it needs to ensure that the integrity tree accurately reflects the state of the data its authenticating. On a re-shaping operation, the Huffmanized Merkle Tree should not create paths through the integrity tree that are inconsistent with the data itself. Otherwise, the hash of some data may not be consistent with its parent, and the hardware will not be able to distinguish whether or not this is due to a corruption or due to a re-shaping. As such, it's imperative that re-shaping the Huffmanized Merkle Tree is a lightweight operation, or else it will become the execution bottleneck. Furthermore, metadata in secure memory protocols typically conforms to very particular layout, either in the traditional secure memory design [35] or in the SGX-style integrity tree design [38], [69]. In both of these designs, the tree is *not* pointer-chasing – the parent of a particular node in the tree is known based on an arithmetic function performed on the address of that node. As such, implementing the Huffmanized Merkle Tree requires lightweight modifications to the traditional layout of secure memory metadata without breaking the cryptographic motivation for their prior layout.

I make the following contributions:

1) I present an argument that an over-reliance on the metadata cache towards optimizing performance in secure memory may lead to unforeseen future security consequences.
2) I propose the Huffmanized Merkle Tree, a novel frequency-based secure memory optimization that is designed to achieve high performance secure memory while *minimizing* the amount of sensitive secure memory metadata stored on-chip.
3) I implement and evaluate the Huffmanized Merkle Tree. In particular, the Huffmanized Merkle Tree provides performance improvements of up to 50% over the baseline secure memory protocol without metadata caches.

In this section, we describe the Huffmanized Merkle Tree. In particular, we will go into the low-level details of the structure, and describe which traditional secure memory semantics need to be adjusted in order to accommodate the Huffmanized Merkle Tree. However, there are a few invariant assumptions in traditional secure memory that our design *cannot* violate. They are:

1) Integrity tree nodes (including counters) should be word aligned (i.e., 64B) for efficient fetches.
2) All data addresses must have an associated major and minor counter to be used for encryption and decryption.
3) The major and minor counters must be in the same state on encrypting some data as they are on decrypting (i.e., the same metadata must be used at all times).
4) Counter values are members of an integrity tree, through which the integrity of all nodes in the tree are protected by a hash function, and the result of that hash is stored in the parent node.
5) The only stored values that can be trusted are those who can be verified against an on-chip value.

In our Huffmanized Merkle Tree, we preserve each of these invariants explicitly. However, we modify the contents of a node in the Huffmanized Merkle Tree such that the tree can be *pointer-chasing*. That is, the address of a parent node in the tree is stored in the node itself rather than being stored at a pre-defined location based on the address of the children. In the process, we achieve an integrity tree design in which the authentication path for a particular node in the tree is provably-optimally short relative to the "hotness" of its access rate.

1) Counters and Tree Nodes: In our design, we must ensure that all secure memory metadata associated with data remains in a word-aligned 64B structure for efficient fetches and conform to design constraint 1. We need to store the pointer to the parent address in the node, which means that we need to reduce the space allocated for other metadata fields in each tree node to make space for the parent pointer. Note, by this we intentionally exclude data from these tree structure semantics, although it can semantically be considered part of the in a traditional setup. This is an intentional design decision that comes from the fact that data should not be modified. As such, the major and minor encryption counters associated with a piece of data are at deterministic addresses that can be mapped directly to the address of the data.

The parents of counters, however, do not need to be in a deterministic location in order to provide integrity. As long as the parent of a counter block stores the hash of that counter (or a hash associated with that counter) then it can provide integrity over that block regardless of where it is located. Fig. 16 demonstrates the new data layout of different nodes types in the pointer-chasing integrity tree. For counters (i.e., leaves), we create space in the node to store the parent pointer by reducing the minor counter size from seven bits to six bits for each of the 64 minor counters. The parent pointer can now be stored in the 64 least significant bits of the block. Inner nodes are typically composed of eight 8-byte hashes (one per child) in traditional integrity trees or eight 7-byte nonces (one per child) and one 8-byte hash of the nonces in the SGX-style integrity tree. In the traditional integrity tree, we reduce the hashes from 8-byte hashes to 7-byte hashes to create a distinct 8-byte field for the parent pointer. In the SGX-style integrity tree, we reduce the nonces from 7-byte counters to 6-byte counters to create the space for the parent pointer.

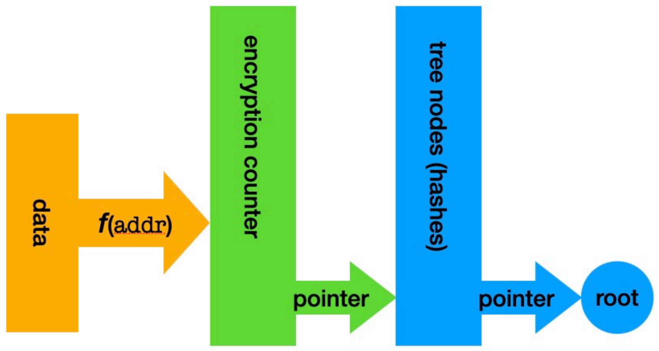


Fig. 16: Depiction of how various types in memory can find their parent

Note, in this setup we allocate 64 bits in order to over approximate the number of addresses of each of the possible metadata in the integrity tree so that the approach can scale for large memories. In practice, for an 8GB there are only 37449 inner nodes (non-counters, as they will only be the direct parent of data) in the integrity tree which requires 18 bits of space. If the amount of space is small relative to the hashes such that the likelihood of collisions is tolerably low, these bits can come from the hash fields as necessary in the SGX-style integrity tree to further reduce the likelihood of an overflow in a nonce.

2) The Huffmanized Merkle Tree: The Huffmanized Merkle Tree is a pointer-chasing integrity tree in which the authentication path length for some data in memory is provably-optimally short relative to its "hotness." By that, we mean that if some data d is accessed n times more frequently than some other data e, the path length through the tree to authenticate d should be shorter than e by optimal factor. Fortunately, the Huffman tree data structure [110] is a well understood data structure that describes the optimal setup for shortest path length, so our integrity tree should conform to Huffman semantics.

In order to ensure that all data values can be authenticated by a counter that can be found reliably, we exclude counters from the Huffmanization process. That is, the counter level of the integrity tree is "sticky," and the major-minor counter for a particular data can be found by computing a function on the address, as is done in the traditional integrity tree setup. In doing so, we satisfy our second and third Huffmanized Merkle Tree design constraint.

However, the parents of counters are determined by following a pointer, and may change over time. As such, whenever the Huffmanized Merkle Tree decides to re-organize itself in accordance with the distribution of accesses, the ancestral path through the integrity tree from leaf to root may actively be changing, so the update of hash value and pointer must appear atomic in the verification process. This is also important to conform to the fourth design constraint. We achieve this by blocking all data authentications while the shape of the Huffmanized Merkle Tree is being adjusted in accordance with the Huffman algorithm. When the shape of the tree is being readjusted in a traditional Merkle Tree, it is important that the hashes and pointers are both adjusted before the value is written back to the memory state. Similarly, in the SGX-style integrity tree, the nonces associated with each child must change along with the pointer upwards. In re-arranging the nonces, this also means updating the hash of the nonces needs to change on writing the value back to memory.

Beyond just conforming to the design constraints of a Huffmanized Merkle Tree, there are several properties that would be beneficial to achieve. In particular, prior art [22] has described a system where frequency counters are incremented on writes to memory. Ideally, we would expand this notion of access frequency to include all accesses, including reads to memory, while not interfering with the critical path of the read operation. This means that using encryption counters as frequency counters is insufficient for a few reasons: (1) they are only incremented on writes to memory, and (2) major counters only describe the number of overflows to minor counters, and *all* minor counters are reset on the increment of a major counter. Consider the case where we use encryption counters for tracking frequency. One possible solution is to use the major counters as frequency counters. However, these update at a relatively slow rate (every 2^6 writes to a word in our scheme), and as such do not provide a granularity fine enough to describe the frequency of access to memory. Another possibility is to use the sum of minor counters, and add to them the major counter $m * 2^6$. This solution, however, is also not entirely representative of the true distribution of access, as an increment to a major counter is due to a single minor counter overflowing. Thus, the other minor counters may all be zero or at maximum value, and using this methodology to track frequency of access similarly loses precision in the frequency granularity. Finally, and most importantly, actively tracking frequency in the integrity tree nodes requires tracking frequency on the critical path of authenticating a node.

As such, it is both cleaner and more performant to use some other counter auxiliary to the encryption counter for tracking frequency of access. The spatial overhead of the secure memory in a traditional integrity tree is approximately 13%, where 12.5% of the space is required by the data HMACs, and less than one percent of the spatial overhead is due to the integrity tree. Adding auxiliary 64-byte frequency counters to this setup is equivalent to doubling the space required to store the encryption counter parents. As such, this has a relatively neglible impact on the spatial overhead (.003%). Thus, in our design, we track auxiliary counters for maintaining frequency for all Huffman tree leaves, and increment them on both reads and writebacks. To keep reads fast, we increment the frequency counter for a particular address after the result of the read has been authenticated and sent to the processor chip.

3) Huffman Algorithm: There are several low-level algorithms that can be used to implement a Huffman trees. We will now describe our methodology for how and why we selected the our approach.

a) Traditional Huffman Trees: One of the original descriptions of the Huffman tree algorithm [111] describes an "ordered linked structure," or priority queue where the head of the queue is the minimum frequency element to build a binary Huffmanized tree. It requires having all of the possible "leaves" in a priority queue at the initialization of the structure. Then, pop the first two items from the queue (i.e., the smallest two elements according to frequency of access) and create a new node which is to be the parent of these two nodes. The frequency of the parent should be the sum of the two children nodes, and its left and right children should be the returned node from the first and second pop operations respectively. Repeat this process until there is only one node in the queue, and that node is the root of the tree.

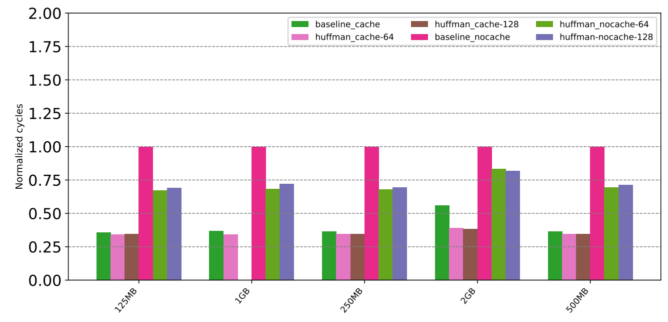
Upon consideration of this algorithm, there are several practical considerations that make it sub-optimal for our approach. For one, we need to block the authentication of data upon updating the shape of the Huffman tree. If we wanted the shape of the tree to be reflective of the current access patterns of the tree, then such an operation could not be infrequent or else the tree would reflect some stale distribution of access. However, actually implementing this algorithm in hardware requires an extreme number of memory accesses. Consider the construction of the priority queue. There are a few means of actually implementing the queue, but likely this queue needs to be pointer-chasing to be able to support re-insertions. As such, searching the queue would be $O(n)$ to perform where each node traversal in the search requires a memory access due to the pointer-chasing nature. There may be complexity class optimizations, like having the queue implement a skip-list [112], that can help reduce this to $O(\log n)$, but this comes at the cost of $2X$ more space to store the priority queue. All of this comes before the construction of the Huffman tree from the priority queue. In total, assuming that you could store eight priority queue nodes in a single 64B word and use all eight of them on a single fetch, this requires 2^{32} memory accesses to construct the priority queue of encryption counter parents in an 8GB secure memory. That is, there are as many accesses required as there are bytes in the memory module, which is unreasonably long for the system to block all main memory accesses.

b) Dynamic Huffman Trees: An alternative Huffman algorithm that has several beneficial properties for the Huffmanized Merkle Tree is the Faller-Gallagher-Knuth (FGK) algorithm [113]. In particular, FGK describes an algorithm in which neither the set of tokens nor frequency of a particular token in a vocabulary is known in advance, and the tree is constructed on-the-fly. For this reason, we describe this algorithm as a "dynamic" Huffman tree algorithm.

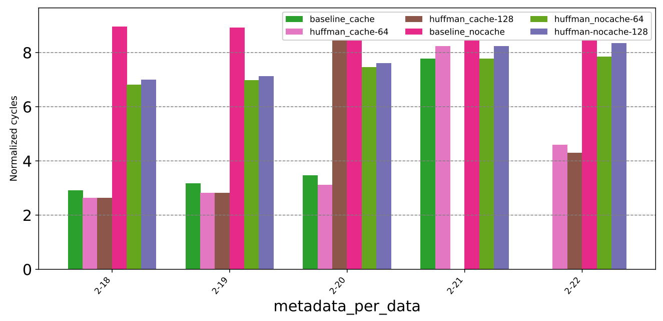
Nodes in the FGK algorithm are classified in terms of *node classes*. Two nodes are in the same node class if they have the same frequency and are either both leaves or both non-leaves. That is, if nodes n_1 and n_2 have the same frequency but n_1 is a leaf and n_2 is the parent of some leaf node, then n_1 and n_2 are in different node classes. However, if n_1 was an inner node or if n_2 was a leaf, then they would belong to the same node class. When some leaf node is accessed, its frequency is increased which results in a re-organization of the tree shape.

The key observation that drives the FGK algorithm is that the ordering of nodes within a node class does not impact the overall skewedness of the Huffman tree. As such, incrementing the frequency of a leaf node in the FGK algorithm results in an "interchange" operation; the node is swapped with rightmost node (where a level closer to the root is "to the right" of a node at a level near the leaves) in the node class. Then, the frequency of the node can be incremented. This either results in promotion to the next node class or creates an entirely new node class. After interchanging and incrementing the weight of the leaf, repeat the process recursively with the parent node until the root of the tree is reached. This process of incrementing a leaf's frequency and restructuring the shape of the tree is a lightweight process compared to the algorithms described above. In particular, we found that updates to the tree shape on restructure are on the order of tens to hundreds of memory operations, which several orders of magnitude less than the standard Huffman tree.

There are a few key distinctions in setup between the Huffmanized Merkle Tree and the original description of the FGK algorithm that require non-trivial design decisions. For one, unlike the FGK setup, we know all possible "tokens" (i.e., data addresses) in the "vocabulary" (i.e., range of memory) a priori. Thus, we don't need to worry about the case of adding new nodes or the zero nodes to the set of leaves, and *must* initialize all leaves in the tree in advance so that all data can be authenticated. This begs the question of how to initialize leaves in the tree. Seeing as a frequency of zero in the root doesn't make sense, we choose to initialize all leaves in the tree with a frequency of one. However, this proves to be problematic. The original description of the FGK algorithm assumed a binary tree, and so the frequency of the parent of any two nodes is the sum of the frequencies of its children. In our case, this implies that the parents of leaves have a starting frequency of eight (as secure memory integrity trees are 8-ary). This implies that, in order for a node to be promoted to the next node class, it needs to have its frequency updated eight times before it is eventually promoted; such a tree has very little restructuring. To



(a) Performance of the Huffman and baseline secure memory with and without metadata cache in graph500.



(b) Metadata per data authentication for parsec benchmarks.

address this, in our implementation, we do not assume that the frequency of a parent node needs to be the sum of its children's frequency. As such, leaves are initialized to have a frequency of one, their parents have a frequency of two, their grandparents have a frequency of four, etc. to resemble how frequencies in the binary FGK tree would look.

Finally, the Huffmanized Merkle Tree needs to block all data accesses on a restructuring operation in order to conform to the fourth design constraint. This is at odds with the fact that the FGK algorithm results in a restructuring of the tree on every memory access, or else the Huffmanized Merkle Tree cannot benefit from parallelism [29]. To achieve both, our implementation of the FGK algorithm in the Huffmanized Merkle Tree utilizes an *update-after* frequency value n. The update-after value implies that the frequency of a particular node in the tree should only be updated after n accesses to that node. Then, upon reaching the interval, the frequency of that node is updated n times to avoid having difficult barriers to reaching the next node class. We found that 64 and 128 make good choices for values of n.

A. Methodology

We implemented and evaluated the Huffmanized Merkle Tree in gem5 [114] full system mode, a state-of-the-art cycle-accurate architecture simulator. In particular, we conduct our evaluation with the hardware described in Table 18. That is, we run with a four-core simulation where each core has private L1 and L2 caches, and a shared L3. Our evaluation is run with Linux v4.14.134, for which there is gem5 long-term support. Within this on-chip configuration, we evaluation the Huffmanized Merkle Tree against two baselines. One baseline has a 16kB metadata cache, and the other has no on-chip resources. Both baselines utilize the standard integrity tree approach (i.e., not pointer-chasing). We then implement and evaluate the FGK algorithm in the secure memory hardware, and reconstruct the tree after every 64 or 128 accesses to a Huffman leaf node, which we specify in the result. We run our evaluation on three different benchmark suites: (1) graph500 [115], (2) PARSEC [116], and (3) SPEC CPU 2017 [95]. These benchmarks all exhibit different properties and are representative of a wide array of workloads.

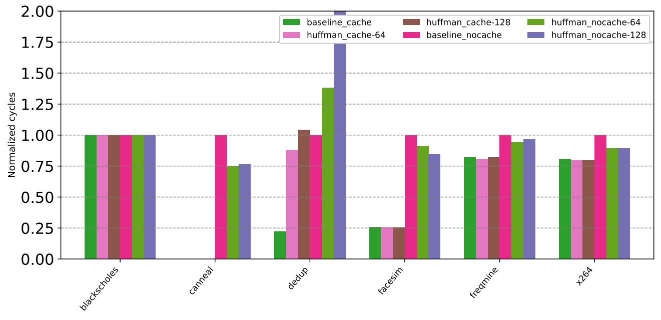
Fig. 18: Evaluation framework configuration.

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	16kB, 2-cycle latency
Huffman Configuration	{64, 128} access per FGK update R/W included in frequency tracking
DDR-based DRAM Configuration	
Capacity	8GB DDR4

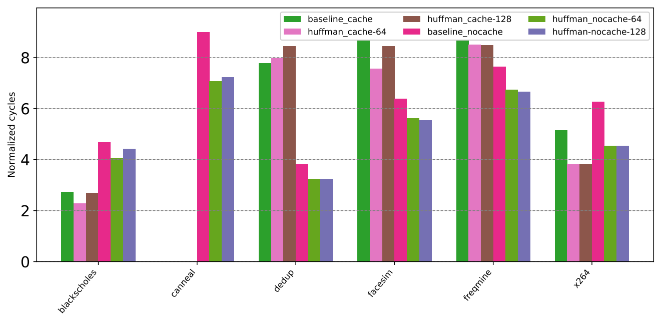
In addition to our simulated evaluation, we also perform local analysis of how these protocols perform based on local memory traces of each of these benchmarks. Under the same simulation specifications, we save the main memory addresses of each of the SPEC CPU 2017 benchmarks to a file, and implement these algorithms locally to perform sensitivity analyses on their respective hyper-parameters. In doing so, we hope to shed light on the nuances and design decisions that influenced the Huffmanized Merkle Tree protocols.

B. graph500 Evaluation

We perform an analysis of the Huffmanized Merkle Tree on the graph500 benchmark. The graph500 benchmark creates a graph of 2^n nodes, with e edges per node and then performs several iterations of breadth first search for particular keys in the graph. It achieves parallelism across cores by running with OpenMPI [117], in which we specify running with four cores explicitly. We disable validation of searches in our evaluation to remain consistent with prior art exploring graph500 [118]. In



(a) Performance of the Huffman and baseline secure memory with and without metadata cache in parsec.



(b) Metadata per data authentication for parsec benchmarks.

our evaluation, we configure the graph with varying sizes of n from $[18, 22]$ with 16 edges per node. This results in graphs with sizes of approximately 125MB, 250MB, 500MB, 1GB, and 2GB for each of the configurations, which we found by running the `mprofile` profiling tool and taking the peak memory consumption. We label the beginning and end of the region of interest in the execution of these benchmarks as being the eighth through twelfth iterations of the breadth first search to exclude graph generation and capture several searches in our measurement.

As demonstrated by Fig. 17a, we find that the Huffmanized Merkle Tree can provide performance improvements of up to 49% relative to the baseline secure memory configuration without a metadata cache in the case of 2^{18} nodes in the graph. As the graph grows in size, the benefits of the Huffmanized Merkle Tree are less apparent. This result makes sense, as intuitively a Huffman tree has more shorter paths if a small number of addresses dominate the relative frequency of access. As more addresses are more frequently accessed, there are more paths through the Huffman tree that will achieve "shorter" paths, but these paths will not be short relative to one another.

To support this assertion, we demonstrate the impact of the Huffmanized Merkle Tree on the total number of metadata per data required to authenticate some data on average, which show in Fig. 17b. Metadata per data implies the number of metadata requests required to authenticate data reads against the integrity tree, and accesses required to update the integrity tree state (which requires a read/update/modify protocol). However, not all metadata requests need to go all the way to the integrity tree root, as parallel requests for similar ancestral paths through the integrity tree can be buffered together as described in [29]. As demonstrated by the figure, the number of metadata required per data access increases from 6.81 in the 125MB graph to 7.84 in the 2GB graph when updating the Huffman tree on every 64 accesses to an address. This increase is significant, as it means that the average work per node is increased.

The other takeaway from these results on the graph500 benchmark is the relative difference in performance between using a metadata cache and not using a metadata cache. In particular, we see that performance is improved by almost $2X$ between the baseline secure memory protocol with a metadata cache versus the Huffmanized Merkle Tree without a metadata cache across configurations. As described in Sec. II, the benefits of using a metadata cache are two-fold: (1) recently accessed metadata values can benefit from faster access if they reside in the metadata cache, and (2) values in the metadata cache reside on-chip, so they are trusted. This implies that authentications can stop early on a metadata cache hit.

The metadata cache in the baseline secure memory case similarly reduces the metadata per data, a la the Huffmanized Merkle Tree. In this case, the baseline secure memory protocol with a metadata cache uses 2.92 metadata per data access. This is much less than the Huffmanized Merkle Tree, and can be largely attributed to strong metadata cache locality. In the 125MB graph, the metadata cache had a hit rate of 79%, which means that many authentications could stop early.

C. *PARSEC Evaluation*

In additional, we evaluate the Huffmanized Merkle Tree against the PARSEC 3.0 benchmark suite [116]. This suite describes several parallel benchmarks with labeled regions of interest. In our evaluation, we measure exactly the specified region of interest. This suite also provides several input sizes for various benchmarks. Our evaluation uses `simlarge`, which is intended to be large enough to stress a simulated environment in a reasonable manner from the perspective of the host.

Fig. 19a demonstrates the performance of the Huffmanized Merkle Tree under the PARSEC benchmark suite relative to the baseline secure memory protocols. In particular, the Huffmanized Merkle Tree performs particularly well in the *canneal* benchmark (33% speedup versus the baseline secure memory protocol without a metadata cache), which is described in the documentation as being adversarial to caching techniques [116]. In other benchmarks, like *dedup*, we find that the Huffmanized Merkle Tree incurs further performance overheads relative to the baseline approach. The baseline protocol with a metadata cache performs well in the *dedup* benchmark, with metadata cache hit rates near 95%.

Metadata per data is reduced in the *dedup* benchmark from 3.82 metadata per data in the baseline secure memory protocol without a metadata cache to 3.23 in the Huffmanized Merkle Tree that restructures on every 64 accesses to an address and

3.25 in the structuring on every 128 accesses for authentication (as per Fig. 19b). Also evident in this particular benchmark was the fact that 27% of all metadata accesses were in Huffman tree reconstruction in the update after 64 accesses per address configuration. For perspective, in the graph500 configuration with 2^{18} nodes, less than 9% of all metadata accesses were part of tree reconstruction. However, when reducing the frequency of Huffman requests to update the tree shape after 128 accesses to an address, the overhead increases. While the number of metadata accesses due to Huffman tree restructuring is reduced to 12% of total metadata requests, the tree is now much less representative of the true distribution, which means that paths that might be accessed soon are on longer authentication paths.

This phenomenon implies that not all workloads are well suited to the Huffmanized Merkle Tree, but it is worth mentioning that many applications and use cases are amenable to it. On average, the Huffmanized Merkle Tree reduces overhead by over 5% relative to the baseline secure memory protocol in four of the six benchmarks in the suite.

D. SPEC CPU 2017 Evaluation

We also evaluate the Huffmanized Merkle Tree under the SPEC CPU 2017 benchmark suite [95]. SPEC CPU is used in several prior arts in the secure memory literature [37], [27], [20], [30], [22]. We run full system simulations in SPEC using gem5, and run local sensitivity analyses based on memory traces produced in gem5 when running these benchmarks, which we describe in Sec. IV-E. Our simulations are run from regions of interest as determined by SimPoint [119], a tool that makes this determination from microarchitectural features of an execution at different intervals. We then evaluate these benchmarks for 500 million instructions from the region of interest.

As demonstrated by Fig. 20, we see that the Huffmanized Merkle Tree improves application throughput in all but one benchmark (*lbm*) with an update interval set to 128 accesses per address (four benchmarks with 64 accesses per address). *lbm* is a highly write-intensive workload, in that 98% of data accesses to main memory in the region of interest come from dirty writebacks from the LLC. As such, it is unlikely that these addresses will exhibit much temporal reuse locality, as they had just been selected as eviction targets from the LLC and the Huffmanized Merkle Tree suffers for performance under *lbm* as a result. Other memory intensive benchmarks, like *mcf*, which exhibit more memory reads (i.e., 65% of data accesses to main memory are reads) have much stronger performance, as the Huffmanized Merkle Tree does not spend time restructuring itself with addresses that are unlikely to be reused. In this case, the Huffmanized Merkle Tree outperforms the baseline secure memory protocol by 16% without a metadata cache.

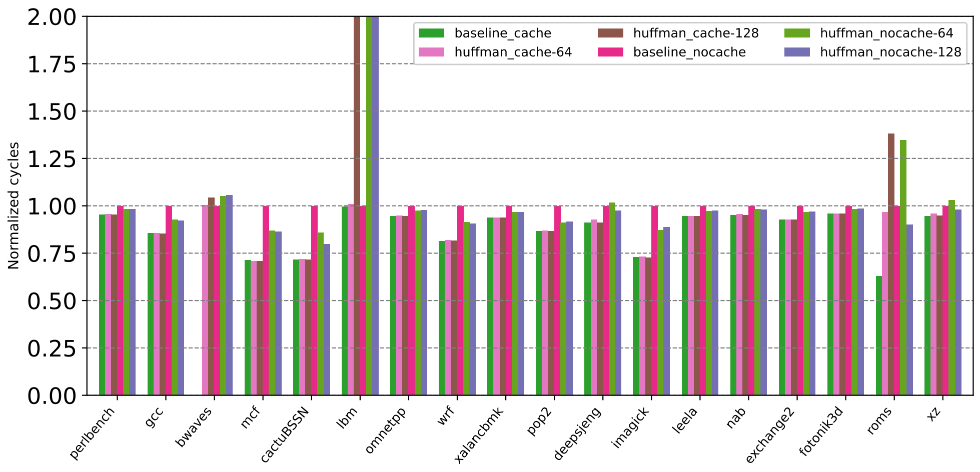


Fig. 20: Performance of the Huffman and baseline secure memory with and without metadata cache in SPEC.

E. Sensitivity Evaluation

V. Baobab Merkle Tree

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 [2]. 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* [120]. Secure memory is defined by a protocol that makes use of a Bonsai Merkle Tree (Bonsai MT) [80]. 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 secure hashes of data. To authenticate data coming from outside the trusted boundary (the chip), the data block's decryption counter is fetched and the counter's integrity is verified against the Bonsai MT by traversing it all the way up to the root, which is stored on-chip and thus its value is trusted. In addition, the data's integrity is verified against its previously stored 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) [121], [37], [122], [31], [123], there has been a lot less work towards resolving (2) [36], [49], [35].

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 similarly 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 section, I present the following contributions:

1) I describe 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) I define a technique to memoize encryption counters on-chip.
3) I evaluate the Baobab Merkle Tree in gem5 [114], and discuss the implications of its design trade-offs.

A. 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.

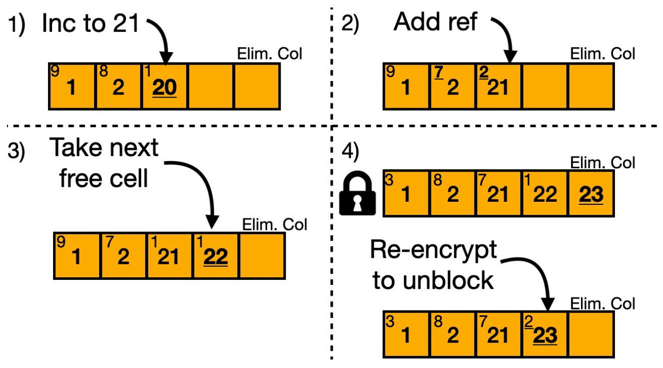


Fig. 21: Incrementing counters using the elimination column.

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. V-A3), the Baobab system needs to consider the number of blocks currently using said counter value. Thus, the proposed design includes a reference counter to track the number of blocks actively using the encryption counter value. Only the $64 - n$ bit counter is used for encryption/decryption, not the reference counter.

The remaining n bits of the column values are used to keep track of the number of blocks currently using the counter. These bits represent a "sticky counter" [124], 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.

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) 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 scenarios in the memoization table: (1) in-place increment (2) next-cell increment, (3) free-cell increment, and (4) blocking increment. We use Fig. 21 to demonstrate each case.

In-place increment occurs when the block that requires incrementing the counter is the only block using that cell (Fig. 21, scenario 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.

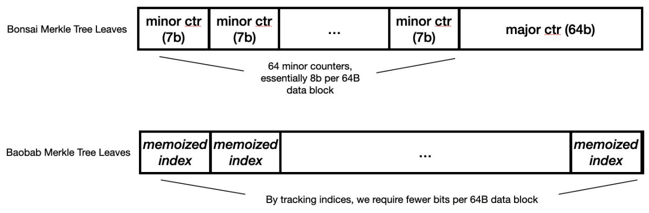

Bonsai Merkle Tree Leaves
minor ctr (7b)
minor ctr (7b)
...
minor ctr (7b)
major ctr (64b)
64 minor counters, essentially 8b per 64B data block
Baobab Merkle Tree Leaves
memoized index
memoized index
...
memoized index
By tracking indices, we require fewer bits per 64B data block

Fig. 22: Layout of data in Baobab Merkle Tree leaves. Fewer bits are required for tracking the index of a cell in a memoization table entry than for major/minor counters.

Next-cell increment (scenario 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 (scenario 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 (scenario 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. 21 scenario 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 re-encrypting those data with the new counter value created in the elimination column (i.e., re-encrypted 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 re-encrypted. To ensure that there is adequate hardware, while re-encryption is happening we block all authentications that require this memoization row.

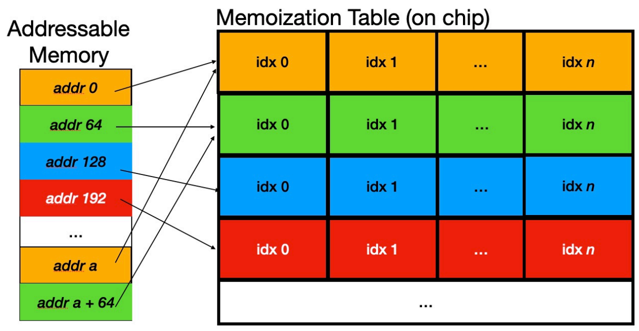


Fig. 23: Memory assignment from address to memoization table row.

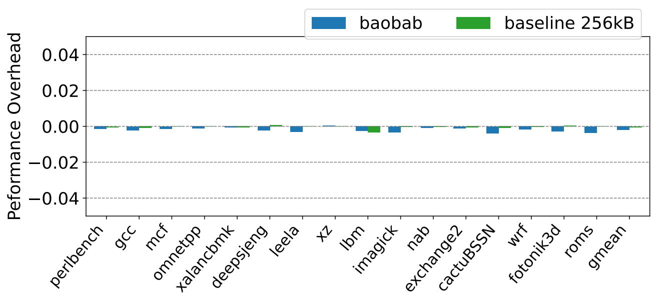
4) Assigning Blocks to Memoization Entries: The assignment of data to memoization table entries is an important feature of the Baobab Merkle Tree. To improve effectiveness, each row assignment works from a heuristic to increase the likelihood of in-place increment and decrease the likelihood of needing 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 23.

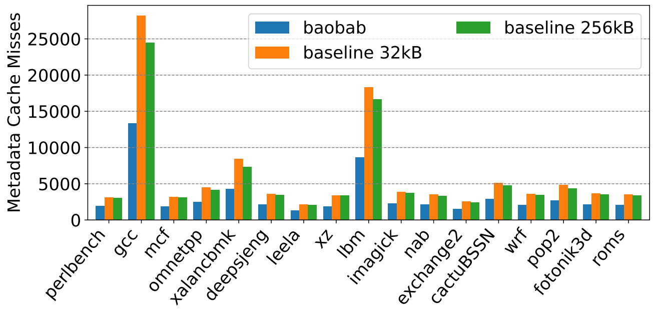
5) Security Implications: In order to uphold secure memory semantics, Bonsai Merkle Trees protect the integrity of encryption counters and use data MACs to ensure that data has not been corrupted [35]. The intuition is that only the untampered encryption counter can produce the decryption key that decrypts 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.

B. Spatial Overhead

The spatial overhead of Merkle Trees in secure memory scales proportionally to the overall memory size. Table 25 shows the amount of reserved memory space required to store the integrity tree. The size of the Baobab Merkle Tree is shown under



(a) Normalized cycles in the SPEC 2017 CPU benchmarks.



(b) Metadata cache misses in SPEC 2017 CPU benchmarks.

Baobab MT, while 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.

C. Runtime Evaluation

We use the SPEC 2017 CPU benchmarks [95] to evaluate the overhead of the Baobab Merkle Tree over a Bonsai Merkle Tree. Fig. 24a 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 [62], the latency to update an memoization table entry is 2ns, which is negligible relative to the memory access latency. For this reason, the overhead due to indirection incurred by the Baobab Merkle Tree is similarly negligible. While there is some additional information being tracked in the memoized data itself (i.e., the sticky reference counters), updating these values can be done at the same cycle and do not incur additional execution overheads. Furthermore, 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 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. 24b 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 a reduction in metadata cache misses.

Fig. 25: 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

VI. Remaining Work

In this section, I will describe the remaining work that I would like to achieve over the duration of my Ph.D. I believe that I can finish all of this work by March of 2025, and am aiming to defend the work pertaining to my dissertation by December of 2024. I have broken this work up into short-term goals (to be done by the end of Fall 2023), intermediate-term goals (to be done by the end of the Spring or during the Summer of 2024), and longer-term goals (to be done over the course of the rest of my Ph.D. and beyond). At the time of writing, I would like to pursue a Post-Doctoral research position after my Ph.D., so further work that isn't done and does not pertain directly towards the defense of my dissertation can continue to be pursued then.

A. Short-Term Goals

At the time of writing, AMNT is currently under submission to HPCA 2024 and we are in the midst of the rebuttal period. The next steps pertaining to that project will be largely determined by the next few days and next week or so. Under the pessimistic assumption that it does not get in, I will spend the next few weeks **polishing** the work and the narrative pertaining to it and plan on resubmitting that work to ASPLOS 2024's fall deadline (November 30th, 2023). This work is the most mature of the work that I have presented in this proposal, and has existed in several different iterations and forms since the end of the Fall in 2020 or the early Spring of 2021, and the feedback in its most recent iteration implies that the work is very close. Once that work has been published, I will turn my attention away from its goals and towards other ends.

Similarly, at the time of writing the Baobab Merkle Tree project is under resubmission to IEEE Computer Architecture Letters, and I anticipate hearing back with regards to its acceptance soon. This project was born out of an idea from a paper in a reading group in December of 2022, and was implemented and evaluated over the course of a few weeks. As such, this project demonstrates the rate at which I anticipate progress to occur in the more mature environment in which I have been developing projects. The next steps in this project depend on its acceptance to IEEE CAL, given that it taught us several things. For one, we learned that indirection is a difficult means by which to achieve performance benefits in a secure memory setting. Furthermore, it taught us that the true savings of the secure memory hardware overheads will come from the data MACs, for which there are 8-bytes stored for every 64-bytes of data. Solving this problem will be difficult, and from it is born an idea that I will describe in long-term goals. However, given these factors, if this work is not accepted to CAL, the short-term goals pertaining to this project likely have to do with **polishing** the work and resubmitting it to a workshop where the techniques will be well received.

Also in the short-term, I would like to get the Huffmanized Merkle Tree project under submission. At the time of writing, the conference that we will target is SPAA 2024, which has a January 24, 2024 deadline. This project proposes interesting ideas with regards to application of data structures in hardware, and has some interesting properties that may be well received at that venue. The work for this project began in the summer of 2022, and the work that I need to do in order to get it in a state where it is submittable is regarding **evaluation** and **writing**. The infrastructure and implementation are solid and done. Seeing as the work challenges semantics of the underlying secure memory protocol's layout at the expense of cryptographic guarantees, and that the project is very much applicable to hardware properties, framing the narrative around this project to suite the venue to which it is submitted will be critical. As such, I anticipate that working towards getting the paper associated with this project accepted will take a few iterations before it is achieved, in which case getting the Huffmanized Merkle Tree project accepted will become an intermediate-term goal.

B. Intermediate-Term Goals

New technologies and architectures have continued to emerge over the course of my Ph.D. One such architecture is Compute-eXpress-Link (CXL) [41]. This architecture promises network speeds that will allow for cross-device coherence [125]. An implication of this new property is that multiple devices will be able to communicate in such a way that a cross-device unified address space is possible. I anticipate that this will have an impact in a variety of areas, and that new programming and consistency models will be born out of literature from RDMA, etc. Of interest to me is the implication of CXL on the underlying secure memory protocol. In particular, I believe that the current literature on distributed secure memory has several short-comings due to a lack of coherent description of assumptions.

Over the course of the spring and into the summer, I would like to take this problem on. In particular, I would like to first spend time **summarizing** the work that exists into several categories of what they are trying to accomplish, and their underlying assumptions. From my brief preliminary exploration into this problem, the current art often begs the question of *if* such a solution is possible, and are often overly complex as a result. However, I believe that this is the wrong approach. Instead, questions should be asked about *how* to best use the additional hardware in the system to optimize the protocol. From this re-framing of the problem, it makes sense to look into which assumptions are and are not reasonable in terms of metadata sharing, and what the baseline protocol should be.

What I have quickly found from mapping out what different protocols might look like is that different models of trust allow for different protocols. For example, in a hyper-paranoid system where no value moving across any network can be trusted, the resulting protocols tend to be boxed into *if* the issue of sharing metadata for remote data is possible (much like the majority of prior art). But this may not be the assumption that makes the most sense for all use cases. For example, a data center storing petabytes of sensitive user data with thousands of employees with varying degrees of physical access to the devices and guest hosts running a wide array of software on the devices dictates a different threat model to a private networked research system at Brown that is locked in a closet. These two applications of secure memory shouldn't necessarily be subject to the same protocol constraints. As such, a goal of mine is to **formalize trust models** in different scenarios and define the parameters that make sense given these trust models with respect to the prior art.

Finally, I would like to **design**, **implement**, and **evaluate** a secure memory metadata coherence mechanism that makes sense for trust models that allow for it. In particular, I believe that in many settings it makes sense for a particular device to authenticate that values have not been corrupted in its local memory before sharing the value remotely. Then, the remote node can trust that the it had been authenticated faithfully and was not modified in transit. As a result, a local node can share values from its metadata cache with remote nodes to store in their metadata caches. Seeing as the remote node will never use this metadata for authentication (each node authenticates its own address space), coherence is intuitive as the remote metadata will only be in a read-only state. If this metadata is found elsewhere via some coherent snoop across the network, then it can be trusted if we assume that secure memory hardware behaves faithfully.

I anticipate that the preliminary research pertaining to this project can be achieved on the order of weeks. From my understanding, this space is still very green and so the papers in this space are relatively limited. Given my existing environment, I similarly anticipate that designing and implementing this protocol will not be an arduous task for more than a few months. Thus, I hope to have this approach evaluated by the beginning of the summer in 2024.

C. Long-Term Goals

After looking into CXL and distributed secure memory, I have found myself interested in the implications on networked systems. It seems like so often the solutions to problems are to throw more and more hardware at a particular system to the point that the performance benefits can be attained, which just seems to be a hugely socially-irresponsible trend in computing from several angles. As such, a long-term goal of mine is to think about problems relating to **socially responsible computing** over the remaining course of my Ph.D. and beyond.

I believe that a hugely under-explored paper is that of Software-Defined Far Memory (SDFM) [33]. The basic premise is to use compression and swap memory to increase the "effective reach" of local memory by compressing some infrequently accessed data. There is an overhead to access some compressed data, by the cost of decompressing it, which is similar to accessing data on some remote node. I think that there are potentially hugely impactful implications of this work on carbon consumption in data centers. If it is the case that SDFM is a viable alternative to getting increased capacity as opposed to accruing more hardware, then this software approach can have positive implications in terms of energy efficiency.

With regards to secure memory, SDFM can be used to reduce the spatial overhead of secure memory. Typically, the majority of secure memory spatial overhead comes from the hashes. However, the hashes do not necessarily need to be the same size for compressed data. That is, hashes only need to represent 64B of data in memory, regardless of whether or not that data is compressed or not. Given this observation, we can use SDFM combined with secure memory to have an effective memory capacity larger than the number of hashes that are protecting it.

VII. Conclusion

In this proposal, I have described how secure memory has reached a new post-SGX age defined by emerging memory architectures. This new wave has been primarily focused on performance as the fundamental challenge, but often fails to address other real challenges. Given the rise of other architecture security questions born out of high performance hardware (i.e., Spectre, etc.), my work in this space brings area overheads and trust into the discussion as first-class design issues. My work AMNT describes a reduced on-chip area overhead for trusted components versus the state-of-the-art by an order of magnitude. My work in the Huffmanized Merkle Tree describes a secure memory protocol that achieves performance without caching. My work in the Baobab Merkle Tree describes a protocol in which the use of on-chip resources are re-examined, and the implications of this on in-memory overhead.

I believe that there is still work to do in this space, and before the completion of my Ph.D. in December 2024 there are several areas that I would like to explore. In particular, I would like to continue my work by considering how trust can impact distributed secure memory. I believe that existing protocols suffer from being overly conservative in order to achieve working protocols, but don't consider how trust impacts their underlying assumptions. Furthermore, I believe that Software-Defined Far Memory has potentially interesting implications on secure memory and beyond in terms of energy efficiently using compute resources and alternative means of increasing system capacity. For each of these, I would like to design, implement, and evaluate the impact of how these re-imagined protocols may affect secure memory from the angles of performance and trust, and extend these to potential implications in socially-responsible compute.

References

[1] U. Otgonbaatar, "Evaluating modern defenses against control flow hijacking," Ph.D. dissertation, Massachusetts Institute of Technology, 2015.

[2] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[3] R. Qiao and M. Seaborn, "A new approach for rowhammer attacks," in *2016 IEEE international symposium on hardware oriented security and trust (HOST)*. IEEE, 2016, pp. 161–166.

[4] O. Mutlu and J. S. Kim, "Rowhammer: A retrospective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 8, pp. 1555–1571, 2019.

[5] Z. Zhang, W. He, Y. Cheng, W. Wang, Y. Gao, D. Liu, K. Li, S. Nepal, A. Fu, and Y. Zou, "Implicit hammer: Cross-privilege-boundary rowhammer through implicit accesses," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[6] A. Kurmus, N. Ioannou, M. Neugschwandtner, N. Papandreou, and T. Parnell, "From random block corruption to privilege escalation: A filesystem attack vector for rowhammer-like attacks," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[7] W. Wu, Y. Chen, X. Xing, and W. Zou, "{KEPLER}: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1187–1204.

[8] M. Seaborn and T. Dullien, "Exploiting the dram rowhammer bug to gain kernel privileges," *Black Hat*, vol. 15, p. 71, 2015.

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

[10] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor, "Checking the correctness of memories," *Algorithmica*, vol. 12, pp. 225–244, 1994.

[11] S. Gueron, "A memory encryption engine suitable for general purpose processors," *Proc. International Association for Cryptologic Research (IACR)*, 2016.

[12] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of the 36th annual international symposium on Computer architecture*, 2009, pp. 2–13.

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

[14] Y. Zhang, L. Gao, J. Yang, X. Zhang, and R. Gupta, "Senss: Security enhancement to symmetric shared memory multiprocessors," in *11th International Symposium on High-Performance Computer Architecture*. IEEE, 2005, pp. 352–362.

[15] R. Elbaz, D. Champagne, R. B. Lee, L. Torres, G. Sassatelli, and P. Guillemin, "Tec-tree: A low-cost, parallelizable tree for efficient defense against memory replay attacks," in *Cryptographic Hardware and Embedded Systems-CHES 2007: 9th International Workshop, Vienna, Austria, September 10-13, 2007. Proceedings 9*. Springer, 2007, pp. 289–302.

[16] G. Duc and R. Keryell, "Cryptopage: An efficient secure architecture with memory encryption, integrity and information leakage protection," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE, 2006, pp. 483–492.

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

[18] Z. Chen, Y. Zhang, and N. Xiao, "Cachetree: Reducing integrity verification overhead of secure nonvolatile memories," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 7, pp. 1340–1353, 2020.

[19] J. Huang and Y. 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.

[20] K. A. Zubair and A. 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.

[21] J. Rakshit and K. Mohanram, "Assure: Authentication scheme for secure energy efficient non-volatile memories," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[22] A. Freij, H. Zhou, and Y. 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.

[23] S. Vig, R. Juneja, and S.-K. Lam, "Dissect: dynamic skew-and-split tree for memory authentication," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 1538–1543.

[24] X. Han, J. Tuck, and A. 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.

[25] S. Liu, K. Seemakhupt, G. Pekhimenko, A. Kolli, and S. Khan, "Janus: Optimizing memory and storage support for non-volatile memory systems," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 143–156.

[26] F. Yang, Y. Lu, Y. Chen, H. Mao, and J. Shu, "No compromises: Secure nvm with crash consistency, write-efficiency and high-performance," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[27] M. Ye, C. Hughes, and A. Awad, "Osiris: A low-cost mechanism to enable restoration of secure non-volatile memories." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2018.

[28] A. Awad, S. Suboh, M. Ye, K. A. Zubair, and M. Al-Wadi, "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.

[29] A. Freij, S. Yuan, H. Zhou, and Y. Solihin, "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.

[30] M. Alwadi, A. Mohaisen, and A. Awad, "Phoenix: Towards persistently secure, recoverable, and nvm friendly tree of counters," *arXiv preprint arXiv:1911.01922*, 2019.

[31] T. S. Lehman, A. D. Hilton, and B. C. Lee, "PoisonIvy: Safe speculation for secure memory," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2016.

[32] F. Yang, Y. Chen, H. Mao, Y. Lu, and J. Shu, "Shieldnvm: An efficient and fast recoverable system for secure non-volatile memory," *ACM Transactions on Storage (TOS)*, vol. 16, no. 2, pp. 1–31, 2020.

[33] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid *et al.*, "Software-defined far memory in warehouse-scale computers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 317–330.

[34] R. Elbaz, D. Champagne, R. Lee, L. Torres, G. Sassatelli, and P. Guillemin, "TEC-Tree: A low cost, parallelizable tree for efficient defense against memory replay attacks," in *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2007.

[35] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," *Proc. International Symposium on Computer Architecture (ISCA)*, 2006.

[36] M. Taassori, A. Shafiee, and R. 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.

[37] A. Awad, M. Ye, Y. Solihin, L. Njilla, and K. A. Zubair, "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.

[38] I. Anati, F. Mckeen, S. Gueron, H. Haitao, S. Johnson, R. Leslie-Hurd, H. Patil, C. Rozas, and H. Shafi, "Intel software guard extensions (Intel SGX)," in *Tutorial at International Symposium on Computer Architecture (ISCA)*, 2015.

[39] P.-L. Aublin, M. Mahhouk, and R. Kapitza, "Towards tees with large secure memory and integrity protection against hw attacks," 2022.

[40] S. Johnson, R. Makaram, A. Santoni, and V. Scarlatta, "Supporting intel sgx on multi-socket platforms." [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/supporting-intel-sgx-on-mulit-socket-platforms.pdf

[41] "CXL:Compute Express Link," https://www.computeexpresslink.org, accessed: 2022-10-14.

[42] M. Alwadi, R. Wang, D. Mohaisen, C. Hughes, S. D. Hammond, and A. Awad, "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.

[43] M. Alshboul, J. Tuck, and Y. 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.

[44] E. Feng, X. Lu, D. Du, B. Yang, X. Jiang, Y. Xia, B. Zang, and H. Chen, "Scalable memory protection in the penglai enclave." in *OSDI*, 2021, pp. 275–294.

[45] Y. Han and J. 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.

[46] X. Han, J. Tuck, and A. 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.

[47] ——, "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.

[48] T. S. Lehman, A. D. Hilton, and B. C. Lee, "MAPS: Understanding metadata access patterns in secure memory," in *Proc. International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2018.

[49] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, J. A. Joao, and M. K. Qureshi, "Morphable counters: Enabling compact integrity trees for low-overhead secure memories," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2018.

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

[51] D. A. McGrew, "Counter mode security: Analysis and recommendations," *Cisco Systems, November*, vol. 2, no. 4, 2002.

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

[53] D. McGrew and J. Viega, "The galois/counter mode of operation (gcm)," *submission to NIST Modes of Operation Process*, vol. 20, pp. 0278–0070, 2004.

[54] J. Yang, L. Gao, and Y. Zhang, "Improving memory encryption performance in secure processors," *IEEE Transactions on Computers*, vol. 54, no. 5, pp. 630–640, 2005.

[55] M. Henson and S. Taylor, "Memory encryption: A survey of existing techniques," *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, pp. 1–26, 2014.

[56] F. Hou and H. He, "Ultra simple way to encrypt non-volatile main memory: Ultra Simple way to encrypt non-volatile main memory," *Security and Communication Networks*, vol. 8, no. 7, pp. 1155–1168, May 2015. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/sec.1071

[57] A. Awad, P. Manadhata, S. Haber, Y. Solihin, and W. Horne, "Silent Shredder: Zero-Cost Shredding for Secure Non-Volatile Main Memory Controllers," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. Atlanta Georgia USA: ACM, Mar. 2016, pp. 263–276. [Online]. Available: https://dl.acm.org/doi/10.1145/2872362.2872377

[58] C. Liu and C. Yang, "Secure and durable (sedura) an integrated encryption and wear-leveling framework for pcm-based main memory," *ACM Sigplan Notices*, vol. 50, no. 5, pp. 1–10, 2015.

[59] S. Swami, J. Rakshit, and K. Mohanram, "SECRET: smartly EnCRypted energy efficient non-volatile memories," in *Proceedings of the 53rd Annual Design Automation Conference*. Austin Texas: ACM, Jun. 2016, pp. 1–6. [Online]. Available: https://dl.acm.org/doi/10.1145/2897937.2898087

[60] S. F. Yitbarek and T. Austin, "Reducing the overhead of authenticated memory encryption using delta encoding and ecc memory," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[61] P. Zuo, Y. Hua, M. Zhao, W. Zhou, and Y. Guo, "Improving the Performance and Endurance of Encrypted Non-Volatile Main Memory through Deduplicating Writes," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Fukuoka: IEEE, Oct. 2018, pp. 442–454. [Online]. Available: https://ieeexplore.ieee.org/document/8574560/

[62] X. Wang, D. Talapkaliyev, M. Hicks, and X. Jian, "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.

[63] M. Haifeng, Y. Nianmin, C. Shaobin, and H. Qilong, "Memory confidentiality and integrity protection method based on variable length counter," *Journal of Algorithms & Computational Technology*, vol. 8, no. 4, pp. 421–439, 2014.

[64] Q. Pei and S. Shin, "Improving the Heavy Re-encryption Overhead of Split Counter Mode Encryption for NVM," in *2021 IEEE 39th International Conference on Computer Design (ICCD)*. Storrs, CT, USA: IEEE, Oct. 2021, pp. 425–432. [Online]. Available: https://ieeexplore.ieee.org/document/9643769/

[65] B. Fitzpatrick, "Distributed caching with memcached," *Linux journal*, vol. 2004, no. 124, p. 5, 2004.

[66] S. Adve and K. Gharachorloo, "Shared memory consistency models: a tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[67] R. Elbaz, D. Champagne, C. Gebotys, R. B. Lee, N. Potlapally, and L. Torres, "Hardware mechanisms for memory authentication: A survey of existing techniques and engines," *Transactions on Computational Science IV: Special Issue on Security in Computing*, pp. 1–22, 2009.

[68] D. Lie, J. Mitchell, C. A. Thekkath, and M. Horowitz, "Specifying and verifying hardware for tamper-resistant software," in *2003 Symposium on Security and Privacy, 2003.* IEEE, 2003, pp. 166–177.

[69] W. E. Hall and C. S. Jutla, "Parallelizable authentication trees," in *Selected Areas in Cryptography: 12th International Workshop, SAC 2005, Kingston, ON, Canada, August 11-12, 2005, Revised Selected Papers 12*. Springer, 2006, pp. 95–109.

[70] K. A. Zubair, S. Gurumurthi, V. Sridharan, and A. Awad, "Soteria: Towards resilient integrity-protected and encrypted non-volatile memories," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1214–1226.

[71] W. Xiong, L. Ke, D. Jankov, M. Kounavis, X. Wang, E. Northup, J. A. Yang, B. Acun, C.-J. Wu, P. T. P. Tang *et al.*, "Secndp: Secure near-data processing with untrusted memory," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 244–258.

[72] C. Nelson, J. Izraelevitz, R. I. Bahar, and T. S. Lehman, "Eliminating micro-architectural side-channel attacks using near memory processing," in *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 2022, pp. 179–189.

[73] B. Rogers, C. Yan, S. Chhabra, M. Prvulovic, and Y. Solihin, "Single-level integrity and confidentiality protection for distributed shared memory multiprocessors," in *Proc. International Symposium on High Performance Computer Architecture (HPCA)*, 2008.

[74] D. Clarke, G. E. Suh, B. Gassend, M. van Dijk, and S. Devadas, "Checking the integrity of memory in a snooping-based symmetric multiprocessor (SMP) system." [Online]. Available: http://csg.csail.mit.edu/pubs/memos/Memo-470/smpMemoryMemo.pdf

[75] O. Shwartz and Y. Birk, "Distributed memory integrity trees," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 159–162, 2018.

[76] M. De Wael, S. Marr, B. De Fraine, T. Van Cutsem, and W. De Meuter, "Partitioned global address space languages," vol. 47, no. 4, pp. 62:1–62:27. [Online]. Available: https://doi.org/10.1145/2716320

[77] S. 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.

[78] J. Yang, Y. Zhang, and L. Gao, "Fast secure processor for inhibiting software piracy and tampering," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2003.

[79] R. C. Merkle, "Protocols for public key cryptosystems." in *Proc. Symposium on Security and Privacy (SP)*, 1980.

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

[81] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, pp. 1–33, 2017.

[82] M. Pendleton, R. Garcia-Lebron, J.-H. Cho, and S. Xu, "A survey on systems security metrics," *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, pp. 1–35, 2016.

[83] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[84] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats." in *USENIX security symposium*, vol. 5, 2005, p. 146.

[85] F. Fleischer, M. Busch, and P. 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.

[86] D. Sgandurra and E. Lupu, "Evolution of attacks, threat models, and solutions for virtualized systems," *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, pp. 1–38, 2016.

[87] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "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.

[88] S. McLaughlin, C. Konstantinou, X. Wang, L. Davi, A.-R. Sadeghi, M. Maniatakos, and R. Karri, "The cybersecurity landscape in industrial control systems," *Proceedings of the IEEE*, vol. 104, no. 5, pp. 1039–1057, 2016.

[89] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 763–780.

[90] I. Oukid, W. Lehner, T. Kissinger, T. Willhalm, and P. Bumbulis, "Instant recovery for main memory databases." in *CIDR*, 2015.

[91] Z. Caklovic, P. Expert, O. Rebholz *et al.*, "Bringing persistent memory technology to sap hana: Opportunities and challenges," *Annual SNIA Persistent Memory Summit*, 2017.

[92] J. Arulraj, A. Pavlo, and S. 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.

[93] M. Lei, F. Li, F. Wang, D. Feng, X. Zou, and R. Xiao, "Secnvm: An efficient and write-friendly metadata crash consistency scheme for secure nvm," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 1, pp. 1–26, 2021.

[94] J. Huang and Y. 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.

[95] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 41–42.

[96] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[97] "Linux kernel documentation," https://www.kernel.org/doc/html/v4.9/kernel-documentation.html, accessed: 2022-07-06.

[98] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

[99] T. Hirofuchi and R. Takano, "A prompt report on the performance of intel optane dc persistent memory module," *IEICE TRANSACTIONS on Information and Systems*, vol. 103, no. 5, pp. 1168–1172, 2020.

[100] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1, pp. 318–319, 2003.

[101] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[102] "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.

[103] V. Costan and S. Devadas, "Intel sgx explained," *Cryptology ePrint Archive*, 2016.

[104] O. Sibert, P. A. Porras, and R. Lindell, "The intel 80/spl times/86 processor architecture: pitfalls for secure systems," in *Proceedings 1995 IEEE Symposium on Security and Privacy*. IEEE, 1995, pp. 211–222.

[105] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020.

[106] Y. Yarom and K. Falkner, "{FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack," in *23rd USENIX security symposium (USENIX security 14)*, 2014, pp. 719–732.

[107] Y. A. Younis, K. Kifayat, Q. Shi, and B. Askwith, "A new prime and probe cache side-channel attack for cloud computing," in *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*. IEEE, 2015, pp. 1718–1724.

[108] R. M. Shadab, Y. Zou, S. Gandham, and M. Lin, "Omt: A run-time adaptive architectural framework for bonsai merkle tree-based secure authentication with embedded heterogeneous memory," in *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2023, pp. 191–202.

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

[110] C. Glassey and R. Karp, "On the optimality of huffman trees," *SIAM Journal on Applied Mathematics*, vol. 31, no. 2, pp. 368–378, 1976.

[111] J. Van Leeuwen, "On the construction of huffman trees." in *ICALP*, 1976, pp. 382–410.

[112] K. Dragicevic and D. Bauer, "A survey of concurrent priority queue algorithms," in *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 2008, pp. 1–6.

[113] J. S. Vitter, "Design and analysis of dynamic huffman codes," *Journal of the ACM (JACM)*, vol. 34, no. 4, pp. 825–845, 1987.

[114] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

[115] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, vol. 19, pp. 45–74, 2010.

[116] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81.

[117] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[118] T. Suzumura, K. Ueno, H. Sato, K. Fujisawa, and S. Matsuoka, "Performance characteristics of graph500 on large-scale distributed environment," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2011, pp. 149–158.

[119] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

[120] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "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.

[121] S. Chhabra and Y. 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.

[122] F. Hou, H. He, N. Xiao, F. Liu, and G. Zhong, "Efficient encryption-authentication of shared bus-memory in smp system," in *International Conference on Computer and Information Technology*, 2010, pp. 871–876.

[123] P. Zuo, Y. Hua, and Y. 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.

[124] R. Shahriyar, S. M. Blackburn, X. Yang, and K. S. McKinley, "Taking off the gloves with reference counting immix," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 93–110, 2013.

[125] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal *et al.*, "Pond: Cxl-based memory pooling systems for cloud platforms," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 574–587.