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

Samuel Thomas
Brown University
USA

Kidus Workneh
University of Colorado, Boulder
USA

Jac McCarty
Bryn Mawr College
USA

Joseph Izraelevitz
University of Colorado, Boulder
USA

Tamara Lehman
University of Colorado, Boulder
USA

R. Iris Bahar
Colorado School of Mines
Brown University
USA

Abstract

Secure memory is a highly desirable property to prevent memory corruption-based attacks. The emergence of non-volatile, storage class memory (SCM) devices presents new challenges for secure memory. Metadata for integrity verification, organized in a Bonsai Merkle Tree (BMT), is cached on-chip in volatile caches, and may be lost on a power failure. As a consequence, care is required to ensure that metadata updates are always propagated into SCM. To optimize metadata updates, state-of-the-art approaches propose lazy update crash consistent metadata schemes. However, few consider the implications of their optimizations on on-chip area, which leads to inefficient utilization of scarce on-chip space. In this paper, we propose *A Midsummer Night's Tree* (AMNT), a novel "tree within a tree" approach to provide crash consistent integrity with low run-time overhead while limiting on-chip area for security metadata. Our approach offloads the potential hardware complexity of our technique to software to keep area overheads low. Our proposed mechanism results in significant improvements (a 41% reduction in execution overhead on average versus the state-of-the-art) for in-memory storage applications while significantly reducing the required on-chip area to implement our protocol.

ACM Reference Format:
Samuel Thomas, Kidus Workneh, Jac McCarty, Joseph Izraelevitz, Tamara Lehman, and R. Iris Bahar. 2024. A Midsummer Night's Tree: Efficient and High Performance Secure SCM. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3620666.3651354

1 Introduction

Traditional systems with volatile memory technology suffer from active and passive physical attacks where data in memory can be selectively targeted and corrupted [33, 45, 48, 55]. Protecting against these vulnerabilities must be done in hardware, and has been thoroughly investigated over the past two decades [20, 38, 78]. However, since volatile memory systems lose their state when power is disconnected, prior work did not have to address the data remanence problem [69]. Storage class memory (SCM) systems use non-volatile technologies as main memory, so data will remain intact even after power is disconnected. Such systems are natural candidates for large memory applications, where there is a lot of data that may be queried and stored. Disk storage is not viable for these cases due to its latency, and volatile memories are not viable due to the lack of persistent data storage. Moreover, SCM systems face new physical attack risks due to their application, providing attackers with extended opportunities for splicing, spoofing, and replay attacks. Thus, SCM requires a solution for protecting data in memory, such as secure memory guarantees and protections.

Standard approaches for providing secure memory for volatile systems use counter-mode encryption to provide data confidentiality and Bonsai Merkle Trees (BMTs) with keyed hash message authentication codes (HMACs) to protect data integrity [47, 58, 71, 80]. The intuition behind these approaches is that the chip's size prevents attackers from inspecting or injecting malicious content, establishing trust for on-chip elements and mistrust for off-chip components. 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 [3, 14, 52], escalate process privileges [18, 24], or any number of other memory corruption-based attacks [23, 46, 63, 64]. SCMs face an additional challenge in that, when an integrity verification fails, it cannot simply reboot the system. 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 [9, 16, 50].

Secure memory systems for volatile memory cannot simply be retrofitted to work with SCM systems as their lack of *crash consistency* means that data cannot be validated across unexpected reboots. An SCM system usually relies on a persistence model to enforce the crash consistency of particular data. The secure memory metadata must similarly 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) and a power outage were to occur, then the persisted data (p_t) would be in an inconsistent state with its corresponding metadata. This means that the secure memory hardware would have no way to distinguish a hash mismatch due to corruption or lack of crash consistency as the metadata would incorrectly 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, as it can lead to steep performance overheads (up to $25X$) at runtime.

An alternative approach, dubbed *leaf metadata persistence* [82], addresses the performance issue by taking a *lazy* approach to crash consistency. That is, only p_t, H_t, and C_t are persisted directly at runtime. The tree nodes T_t are written to the volatile metadata cache and only written back to memory on eviction (i.e., they are not written-through directly). After a crash, at system recovery, each of the inner nodes of the integrity tree are recomputed from the hashes of its leaves (i.e., the counters). If the computed tree root matches the stored tree root, then the system can be safely rebooted. However, this recovery procedure is pessimistic because all inner nodes of the tree are assumed to be stale/untrusted, and recovery will be worse as memory capacities continue to grow beyond the scale of current SCM devices. These two extreme baselines describe an inherent trade-off between runtime *performance overhead* and *recovery time*. That is, performance overhead is reduced as crash consistency models become lazier, but at the cost of increasingly unreasonable recovery times.

The current state-of-the-art navigates this trade-off space, and can largely be categorized as being either a *static* [10, 65, 82] or *dynamic* [4–6, 19, 26, 42, 56, 76, 85] negotiation of performance overhead due to crash consistency and recovery time. Static approaches work well to strictly reduce the overhead due to maintaining the crash consistency of secure memory metadata [82], reducing the work required at recovery time [10], or partitioning hybrid untrusted device semantics [65], but these approaches miss out on potential performance benefits by treating all addresses the same. Dynamic approaches, on the other hand, explicitly track application behavior to ensure that "hot regions" of memory benefit from having shorter paths through the integrity tree to persist values [26, 56], maintain an auxiliary "fast tree" in which frequently accessed values can be directed to a more relaxed crash consistency protocol [6, 76], or add auxiliary structures to further protect the metadata cache [4, 5, 19, 42, 85]. A limitation of these approaches, however, is that hot region tracking in hardware is difficult, and is dependent on application behaviors. Furthermore, these approaches tend to come at the cost of *hardware complexity*, a third component in this trade-off space that hasn't been emphasized to the same extent as performance or recovery.

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

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

AMNT works from the insight that certain "hot" regions of physical memory may be accessed with more regularity, whereas an application may never access other regions. We leverage this insight by implementing a hot-region tracking mechanism in which a small region in memory gets to

[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."

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.

We make the following contributions:

- We present AMNT, a dynamic hybrid metadata persistence scheme for secure SCMs that performs hot region tracking to adapt to in-memory behaviors at runtime.
- We introduce AMNT++, an optional hardware-software co-design physical page allocator that acts as an addition to AMNT in order to improve the likelihood of an in-use page to be tracked in the hot region.
- We show how AMNT uses 96 bytes of volatile on-chip space and 64 bytes of non-volatile on-chip space, which is agnostic to memory and metadata cache size.
- We demonstrate how a system administrator can bound the recovery time using our proposed approach to achieve desired performance goals.

2 Background

Efficient techniques have been researched for designing traditional (volatile) secure main memory. However, new security challenges arise with storage class memory, which also affects runtime performance and recovery.

2.1 Traditional Secure Main Memory

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

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

To provide protection from replay attacks, secure memory systems use a *Bonsai Merkle Tree* (BMT) [58], which is a modified Merkle Tree (a tree of hashes) that protects the integrity of the encryption counters [47, 58] (depicted in Figure 1). The BMT consists of counter hashes (leaves) that are hashed together to form the next-level node (parent). This process is repeated recursively until a single node remains (i.e., the root of the tree). To establish the root of trust, the root of the tree must always be stored on-chip, as this is the trusted boundary. The BMT is an efficient way of constructing and storing an integrity tree as it reduces the tree's memory requirement [7, 8, 28]). Furthermore, the BMT root is small and unique to the state of the entire underlying memory, so it allows for trusted verification with very little on-chip area overhead.

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

To reduce the integrity verification latency, most prior work assumes that security metadata (integrity tree, encryption counters and data HMACs) is kept in on-chip caches [27, 40, 41, 44, 59–61, 66, 67, 70–72, 78–80]. Doing so introduces a two-fold optimization. First, like data caches, recently accessed metadata can be re-accessed with lower access latency. Second, nodes cached on-chip are members of the trusted compute base, so they too can act as roots of trust, reducing the integrity verification path. The implication of this phenomenon is that the performance overhead of secure memory protocols is largely tied to the efficacy of the secure memory metadata cache in that it significantly reduces the number of metadata fetches required to authenticate data in memory. With the growing memory footprint of applications, the secure memory metadata cache hit rate decreases, requiring more memory accesses and tightly binding application performance to the secure memory system.

Bonsai Merkle Trees come in two forms: General BMTs and SGX-style BMTs. General BMTs describe a BMT structure where nodes are the concatenated hashes of their children (like that in Figure 1). SGX-style trees have counters embedded in each node throughout the tree, concatenated with a hash of its parent. In this form the hash is computed using the Galois-counter hashing mode [28] with the counter from the parent. This work assumes the General BMT format. However, the proposed protocol can be used in an SGX-style BMT with small modifications to take advantage of protocols that target this format, such as Anubis [85] and Osiris [82].

2.2 Secure SCM

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

Various software libraries (e.g. [17, 21, 34, 54, 75]) and hardware extensions (e.g. [49, 84]) provide the programmer sufficient control to avoid data inconsistency problems and ensure application data is crash consistent after a power failure. Like application data, security metadata (i.e., HMACs, counters, and the BMT) must also be crash consistent to ensure that data can be authenticated after a crash. Unlike application data, secure memory metadata is not accessible to the application, requiring the on-chip memory controller to enforce its persistence. An implication of this phenomenon is that the root of the BMT must always reflect the state of data in main memory, and updates to the state and the root must be atomic. Furthermore, the root of the BMT must be stored in a non-volatile on-chip register in order to be trusted through a crash.

2.3 Metadata Persistence Strategies

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

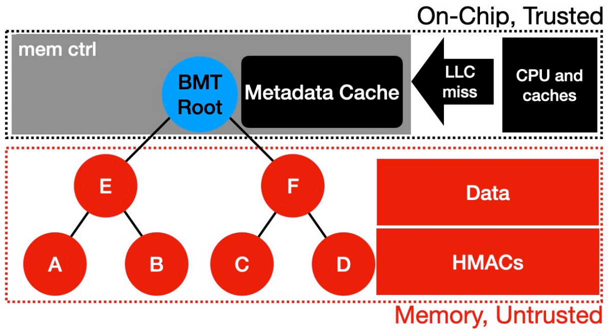


Figure 1. Bonsai Merkle Tree (BMT). Inner tree nodes are the concatenation of the child hashes.

By contrast, a *leaf metadata persistence strategy*, while improving runtime performance, significantly increases recovery time. In this strategy, only the BMT leaf (i.e., counter) and root updates are done atomically with a data write. The rest of the security metadata is updated lazily on a writeback from the metadata cache. On system failure, all inner BMT nodes must be assumed to be stale in SCM, and must be recomputed. In order to recompute the BMT nodes, BMT leaves must be fetched and their hashes computed. Inner-BMT nodes are composed of the keyed hashes of their children, which makes the computation of a node in the BMT dependent on the fetch of each of its children. The data dependent nature of BMTs limits the number of productive parallel memory fetches to BMT sibling nodes, and implies that a large number of bursts (proportional to memory size) must be performed in order to recompute all of the inner BMT nodes. As such, recomputation can last billions of cycles and spans all of secure memory metadata, which, for SCM, may run into the terabytes. Once recomputed, the BMT hashes are compared to the root, which is stored securely and persistently on-chip.

Prior art Osiris [82] further relaxes the leaf metadata persistence protocol by introducing a "stop-loss" persistent metadata cache for BMT leaves. The protocol persists leaves after every n data updates to ensure that they can never be more stale than the stop-loss frequency. As a result of the persistence relaxation, full BMT recovery in Osiris is slower than in leaf metadata persistence. Prior art suggested statically partitioning the persistence policy of metadata based on its tree level [10] or its location in volatile or non-volatile memory [65]. However, to the best of our knowledge, there is no work that proposes a dynamic persistence scheme.

Another state-of-the-art technique, Anubis [85], takes an approach that tracks the addresses currently residing in the metadata cache—only these addresses need to be recomputed at recovery time. It tracks this information by "shadowing" these addresses in a new structure in memory, called the "shadow table." This shadow table tracks updates to cached metadata nodes in order to have a precise view of which metadata are stale. Instead of performing whole tree reconstruction, this protocol corrects only the stale values in memory at recovery time. However, the slow path of this approach occurs on every metadata cache miss, which means its performance is directly tied to an application's metadata cache locality. This technique is also a movement away from reducing the spatial overhead of secure memory [58, 60, 73].

Other prior work, leverage the insight that the length of a persist path may be adjusted in accordance with the relative hotness of some set of addresses. ProMT [6] and Bo-Tree [76] propose using indirection towards a "fast tree" outside of the global tree if addresses are deemed to be hot enough. ASSURE [56] proposes a single Dynamic Modifiable Merkle Tree that changes location based on the hotness of a subtree. Along these lines, Bonsai Merkle Forest [26] reduces the overhead of strict persistence by storing frequently accessed subtree roots in a non-volatile on-chip cache. To determine which nodes qualify, counters are added to the the volatile metadata cache lines, and nodes in the persistent root set (i.e., non-volatile metadata cache) are either "pruned" or "merged" depending on the counter state.

3 Threat Model

Our assumed threat model trusts on-chip values and considers everything else untrusted [43, 71]. The off-chip components (*i.e.* memory, disk, I/O) are vulnerable to passive (snooping) and active (splicing and replay) physical attacks, from both legitimate and illegitimate users. Our proposed work is focused on protecting in-memory data in a system that uses an SCM device. We assume the attacker has physical access to the system and can run any legal program to exploit potential off-chip physical device vulnerabilities. Similar to prior work [5, 25, 26, 82, 85], protection against information leakage via side-channel attacks is out-of-scope for our proposed defense mechanism.

4 A Midsummer Night's Tree

The proposed solution, *A Midsummer Night's Tree* (AMNT), is a secure SCM protocol that balances a reasonable runtime overhead with controllable recovery times and minimal hardware overhead. AMNT achieves its goals by using dynamic hybrid metadata persistence strategies within the same BMT [58].

4.1 "A Tree Within a Tree"

We work from the assumption that a small number of contiguous addresses in physical memory are frequently accessed (i.e., "hot"). Given this assumption, we propose AMNT, which protects a small region of physical memory with a fast persistence protocol while most addresses are persisted strictly

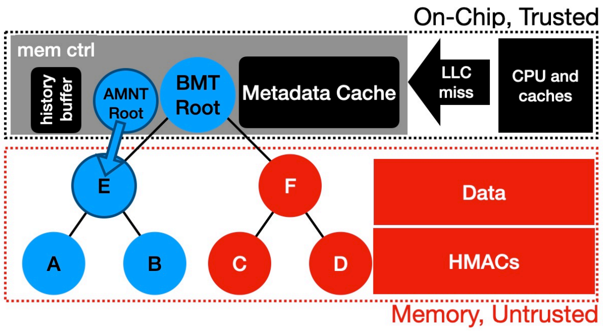


Figure 2. A Midsummer Night's Tree. Red nodes implement strict persistence. Blue nodes implement leaf persistence.

to keep the work required at recovery time low. AMNT is a dynamic metadata persistence protocol that tracks hot regions of physical memory within a *subtree* of the underlying BMT in order to adapt to changing in-memory hotness at runtime. The subtree implements a leaf persistence strategy, where tree nodes are assumed to be stale at the time of a system failure (blue nodes in Figure 2). The rest of the BMT implements strict persistence to minimize recovery time after a crash (red nodes in Figure 2). Given that a small contiguous region of addresses are frequently accessed, we make updating them fast. Implementing strict persistence outside of the subtree, while slow at runtime, will not occur often, minimizing the impact on overall performance and reducing the work required at recovery time.

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

Given the metadata persistence strategy, all values outside the subtree root in the BMT are not stale at the time of a crash. To recover the BMT, AMNT only needs to recompute nodes inside the subtree; recovery time depends on the subtree size determined by the subtree root level. System administrators can control recovery time by configuring the subtree root level in the BIOS, and we offer insight into the trade-off between recovery time and runtime overhead in Section 6.

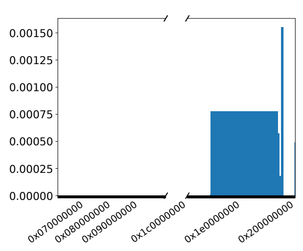
4.2 Hot Region Tracking

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

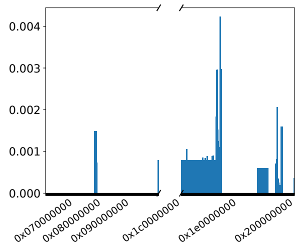
The history buffer has n entries and tracks the n most recent memory writes. Each entry has a subtree index (identified by the index of the node within the subtree level) and a $log_2 n$ counter. On a data write, the subtree index of the corresponding data address is updated by scanning the history buffer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

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

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



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



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

Figure 3. Memory accesses per address in single program and multiprogram workloads.

5 AMNT++

AMNT is predicated on the assumption that a small, contiguous address range in physical memory is protected by the fast subtree. However, while this assumption may hold true for a single application, it may not in a realistic scenario (*i.e.* multiprogram environment). Figure 3a shows the number of memory accesses per physical address in the *lbm* benchmark from the SPEC CPU benchmark suite, whereas Figure 3b shows the number of memory accesses per physical address when running two programs (*perlbench* and *lbm*) in parallel. As evidenced by the figure, multicore systems that run multiple applications prove the assumption driving AMNT may not exist in practice. To address this limitation, we consider a protocol that has "per-core subtrees" to track hotness, but such a solution would result in complex and large hardware requirements for devices with hundreds of cores. Instead, we propose a *hardware-software co-design* to keep hardware complexity low and modify application behavior from the operating system's memory management unit to bias allocation of physical pages towards highly available subtrees in order to maximize potential subtree locality.

To increase the effectiveness of AMNT, all applications should ideally work in the same subtree region whenever possible to increase subtree locality. Consolidating frequent memory accesses into a single subtree can be an important performance optimization, which we attempt to maximize through lightweight modifications to the physical page allocator in the operating system (OS).

In Linux, allocating physical memory is a distinct procedure from allocating memory at the application level. Theoretically, cross-page locality may be unlikely given that physical pages will be allocated according to a *binary buddy allocation* scheme and where "random" pages are reclaimed by the OS over time. This allocation makes it difficult to reason about where two virtual pages are in physical memory relative to each other.

Table 1. System configuration.

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 [35], 391ns write [31]

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

In 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. In the AMNT++ modification, the linked list is reordered to place chunks within the subtree region at the head of the linked list. This approach makes each individual allocation as fast as the standard physical allocator by taking the restructuring of the linked list out of the critical path of a physical allocation.

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

The design decision to bias the allocator towards physical pages at the granularity of subtree regions is intentional. The

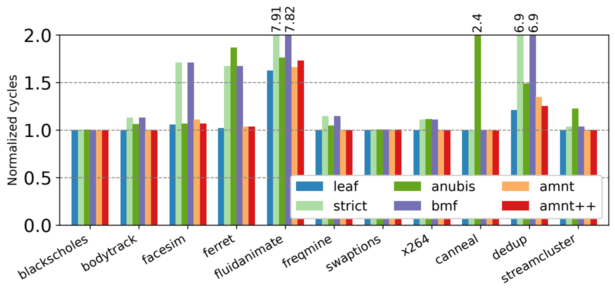


Figure 4. Normalized cycles in single program PARSEC workloads.

AMNT++ protocol biases the `free_list` towards pages in the subtree region, so that the OS does not have to perform these exhaustive searches when allocating pages. Given that the locality of interest is in terms of the integrity tree nodes coverage (e.g. at level 3 the coverage is 128MB for an 8GB memory and scales with increased memory size), achieving this locality is a reasonable task, even in a fragmented system. As a result, the benefit of AMNT++ is high without being overly intrusive on the execution of the OS and application.

6 Evaluation

We first evaluate AMNT on the PARSEC benchmark suite [11] version 3.0 with the simlarge inputs in gem5 [12], a cycle accurate processor simulator. The processor configuration includes a single core with a 32kB data L1 cache, a 48kB instruction L1 cache and 1MB L2 cache. We configure the processor with intentionally small on-chip caches to stress the memory system and show the overheads of the secure memory hardware. The memory system configuration is shown in Table 1. The secure memory hardware includes a 64kB metadata cache with an 8-level BMT to remain consistent with Intel SGX's configuration [29].

We choose to use PARSEC as the primary means of evaluation due to its diversity of workloads and labeling of the beginning and end of the region of interest. The latter is important—determining the region of interest in benchmark suites without labeled regions of interest is typically done by running a profiling tool, like SimPoint [53]. These tools determine the region of interest based on microarchitectural characteristics of the workload, and the application is run from this point for a set number of instructions. However, we cannot compare AMNT and AMNT++ using this methodology as the modified OS results in a different number of instructions, requiring different points in the program to execute the same region of the application.

Throughout the evaluation we compare the proposed approaches (AMNT and AMNT++) against the **leaf** metadata policy and the **strict** persistence policy. In addition, we also compare the proposed approaches against various protocols

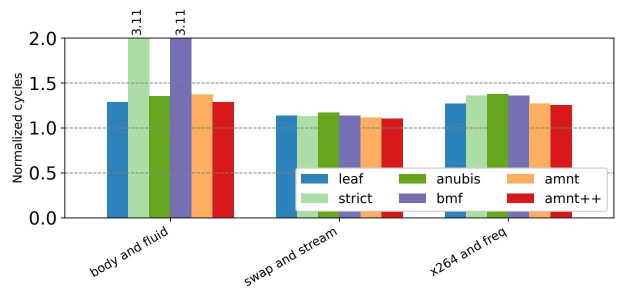


Figure 5. Normalized cycles in multiprogram PARSEC workloads.

proposed in the literature, **anubis** and **bmf**, which we implemented based on their design descriptions [26, 85]. Results labeled as **amnt** show the proposed protocol without the modified operating system, and **amnt++** show AMNT with the modified operating system.

6.1 Single Program Analysis

On average, AMNT has a 16% performance overhead relative to the volatile secure memory scheme and AMNT++ has a 10% performance overhead. Figure 4 shows the cycles for each configuration normalized to the volatile secure memory results. Leaf and strict persistence have 8% and 2.39X performance overhead respectively. AMNT effectively negotiates the trade-off between leaf and strict persistence, achieving its design goal of having the near-leaf performance overhead.

On the other hand, Anubis [85] largely benefits from leaf persistence, but incurs a slow-path case on a metadata cache miss. For workloads that have bad metadata cache efficacy (i.e., *canneal*), Anubis results in large performance overhead, 2.4X compared to the volatile secure memory system. *canneal* has 30.4% metadata cache hit rate. In contrast, AMNT's performance is directly dependent on the application's spatial locality, as opposed to cache efficacy, and as a result is able to lower *canneal*'s overhead down to less than 0.1%.

6.2 Multiprogram Analysis

As described in Section 5, running a single program may not fully stress the underlying protocol in AMNT as the single program's address space will be the only one exhibiting locality in memory. Thus, in this section we use multiprogram workloads to approximate real-world behaviors in which the memory system is subject to the interference due to the interaction of multiple processes. In order to perform such an evaluation, we configure our environment with combinations of programs with temporarily similar regions of interest from the PARSEC benchmark suite. The multiprogram evaluation methodology is consistent with prior work [74]. To choose the pairs of multiprogram workloads, we selected benchmarks whose region of interest appeared at the most similar times to ensure that the regions of interest of each benchmark is evaluated in parallel. These workloads are: *bodytrack* and *fluidanimate*, *swaptions* and *streamcluster*, and *x264* and *freqmine*. To ensure that the evaluation always covers both benchmarks' regions of interest, we start measuring when the second benchmark reaches the beginning of its region of interest, and stop the simulation when the first benchmark reaches the end of its region of interest. The presented results come from running both region of interests in parallel. The simulator configuration for the multiprogram analysis includes two cores, each with a private 32kB data L1 cache, 48kB instruction L1 cache, and 128kB L2 cache. Both cores share a 1MB L3 cache.

AMNT++ is effective at improving AMNT performance. Figure 5 shows the performance normalized to the volatile secure memory setting for all the approaches for the three pairs of multiprogram workloads (including prior work and AMNT and AMNT++). AMNT++ counteracts the multiprogram behavior that impacts the efficacy of AMNT. For example, the *bodytrack* and *fluidanimate* workload display an example when applications may impact their mutual spatial locality. In this scenario, AMNT++ can make a difference by reordering the physical pages accordingly and increasing subtree hit rate from 91% to 97%. As a result, AMNT++ performance overhead is reduced from 8% (AMNT) to less than .1% compared to leaf persistence (the best performing approach). Note that the performance overhead of AMNT in the single program experiments for *bodytrack* and *fluidanimate* are less than .1% and 2.1% respectively.

The *swaptions* and *streamcluster* workload and the *x264* and *freqmine* workload are not memory intensive, and as a result the performance overhead is negligible across both approaches. One could imagine a theoretically adversarial case in which the subtree bounces back and forth between subtree regions, especially with distinct address spaces for multiple processes. Our evaluation finds that these cases do not occur in practice. In the study of single program workloads, we find that the subtree root moves 0.3% of data accesses on average (3 subtree root movements for every 1000 data memory accesses). In multiprogram workloads, this occurs 0.1% of data accesses, which implies no significant difference in memory behavior.

6.3 Subtree Sensitivity Analysis

As described in Section 4, the AMNT subtree level can be configured in the BIOS so that the hardware can be modified for varying workload characteristics. To build an intuition about these workload characteristics, we present a sensitivity study of storing the subtree at varying levels under the multiprogram workloads. As these configurations are intentionally designed to constrain the efficacy of AMNT, we also use this section to demonstrate the efficacy of AMNT++ at changing application behavior to improve subtree hit rates.

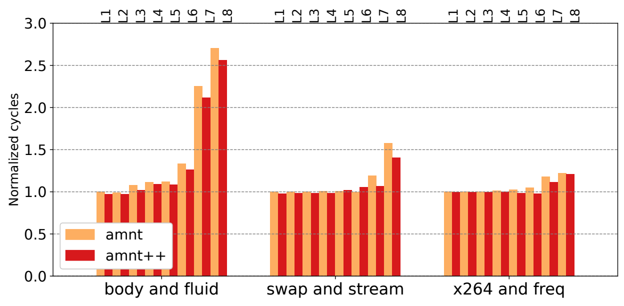


Figure 6. Normalized cycles in multiprogram PARSEC workloads varying AMNT subtree level.

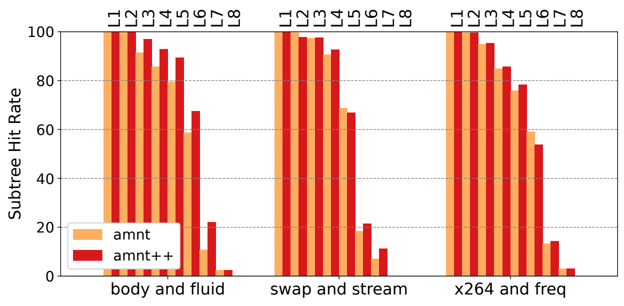


Figure 7. Subtree hit rates for multiprogram PARSEC workloads varying AMNT subtree level.

Figure 6 shows performance impact of varying the AMNT subtree root level with and without the modified operating system (AMNT++ and AMNT), and Figure 7 shows the subtree hit rates for the same configurations. As the subtree root level increases (i.e., moves closer towards the leaves), it protects less data thereby constraining its efficacy. For example, in the *bodytrack and fluidanimate* workload, the subtree hit rate improves by at least 5% when the subtree root is placed between levels 3 and 7 with AMNT++ compared to AMNT. On the other hand, workloads like *swaptions and streamcluster* and *x264 and freqmine*, the runtime performance is not bound by the secure memory hardware and as a result the performance impact is less evident. This result demonstrates that AMNT++ is able to improve the hot region tracking of the underlying hardware without needing to complicate the hardware in a multiprogram environment.

6.4 The Cost of AMNT++

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

Table 2 describes the impact of the modified OS on application behavior. The normalized performance column reflects the number of cycles to run the multiprogram workload with the modified OS over the number of cycles with the unmodified OS. The impact of the modified OS is negligible on the overall performance. This result is due to the fact that the physical reclamation of pages is an infrequent operation, and the modifications are mostly transparent to the progress of the application.

On the other hand, the number of additional instructions of the OS modification is relatively small. The second column of Table 2, the instruction overhead, reflects the additional number of instructions in the modified OS compared to the unmodified OS. On average across the PARSEC benchmark suite, the instruction overhead of AMNT++ is 1.96%.

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

6.5 Multithread Analysis

We evaluate AMNT on the SPEC CPU 2017 benchmark suite [13] to perform multithreaded analysis consistent with prior art [5, 10, 25, 82, 85]. We run the speed benchmarks with ref inputs, and we fast-forward to a region of interest as determined by SimPoint [53] in the benchmark before simulating 500 million instructions. As done in prior work, we use a four core simulation with a 32kB data L1 cache, 48kB instruction L1 cache, 512kB L2 cache, and 8MB L3 cache.

Figure 8 shows the normalized cycles of the SPEC CPU 2017 benchmarks over a volatile secure memory system which does not account for the persistent state of the metadata. AMNT reduces runtime overhead by as much as 41%

Table 2. Impact of the modified operating system in multiprogram workloads.

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

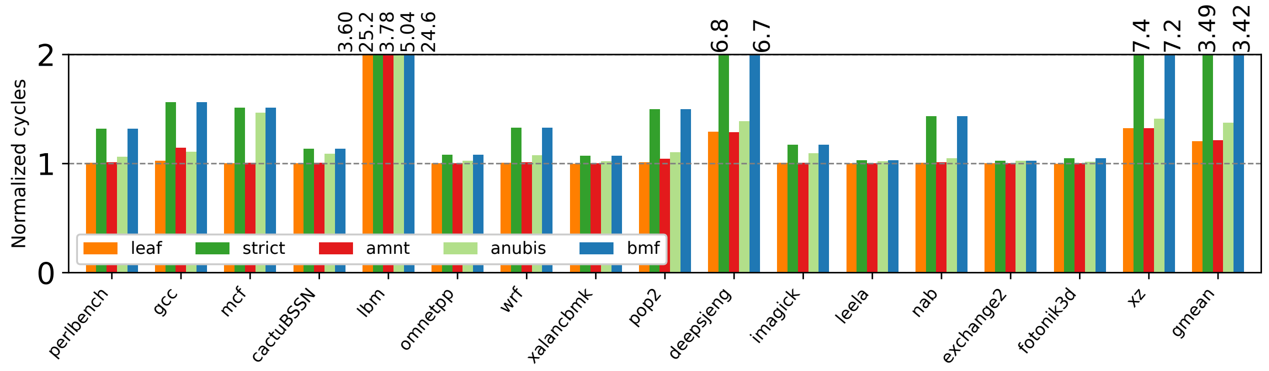


Figure 8. Runtime comparison of AMNT, Anubis, and BMF protocols for the SPEC 2017 CPU benchmarks normalized to writeback secure memory protocol. Lower is better.

and by 13% on average compared to the state-of-the-art, Anubis [85]. Compared to the leaf and strict persistence baselines, AMNT has a runtime overhead of less than 2% compared to leaf persistence, and up to an 8X reduction in overhead relative to strict 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 a $7X$ overhead. AMNT reduces the runtime overhead as it uses leaf persistence semantics on the hot regions of the programs, while keeping the recovery time bounded to a predefined amount. Read-intensive applications are largely optimized by volatile on-chip caches and are unaffected by the metadata persistence model. However, for mechanisms that add complex calculation for memory reads (such as Anubis and BMF), the persistence model still adds to the runtime overhead. For example, AMNT exhibits negligible overhead versus leaf in *cactuBSSN* and *mcf* because they are mostly read memory-intensive benchmarks. Yet, Anubis and BMF both have significant overhead. Anubis suffers from costly metadata cache misses and BMF simply resembles the behavior of the strict performance protocol resulting in high performance overhead. In contrast, AMNT improves performance overhead as it optimizes metadata cache behavior.

6.6 Hardware Overhead

Spatial overheads for secure memory should be minimized as on-chip area is in high demand for various hardware optimizations across a multitude of workloads. For example, if secure memory hardware occupies on-chip space for the LLC, then the application will incur more LLC misses and be further bound by secure memory. Furthermore, applications are becoming more memory intensive, placing more emphasis on the importance of caching values in the larger components of the chip (*i.e.* LLC). Finally, trends in secure memory have moved towards reducing the in-memory spatial overhead of secure memory, so storing more security metadata in memory is undesirable [60, 73, 79].

Table 3. Hardware overheads of the state-of-the-art for a 64kB metadata cache. Note that BMF overheads is metadata cache size dependent and it requires an additional 6 bits of volatile capacity per cache line.

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

We compare the on-chip volatile and non-volatile overheads separately as these may be composed of different technologies (SRAM vs. Flash). The hardware area overheads for Anubis, BMF and AMNT are listed in Table 3. We assume that the on-chip root of the BMT must reside in on-chip Flash to conform to the threat model which keeps the root on-chip at all times for all three mechanisms. In BMF, the non-volatile on-chip space consists of an additional non-volatile metadata cache used to store the subtree roots (4kB by default in that work). The volatile on-chip space requires an additional 6 bits per cache line for the frequency counters in addition to the metadata cache. For a 64kB metadata cache, which holds 1024 64B cache lines, the frequency counters amount to 768B.

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

Area overhead in both of these prior works is a function of memory size. In BMF, a bigger metadata cache will result in more space for frequency counters, or new workloads might demand more subtrees to be tracked for performance.

Table 4. Recovery times (in ms) for the different protocols as a function of memory size.

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

Our implementation of Anubis include the shadow table in a distinct on-chip cache for a 64kB metadata cache to remain consistent with [29], and we model the non-volatile metadata cache in BMF as a distinct on-chip cache with a non-volatile metadata cache proportional to the volatile metadata cache size (4kB).

In contrast, one of AMNT's goals is to limit the additional hardware components both on-chip and in-memory. Like Anubis, AMNT has an additional non-volatile register on-chip to track the root of the fast subtree. The volatile on-chip space includes a 768-bit history buffer, and the metadata cache. The 37kB volatile shadow cache in Anubis is much bigger in capacity than the volatile history buffer required for AMNT. The 4kB non-volatile subtree root cache in BMF requires a significantly greater non-volatile on-chip resource compared to the single non-volatile on-chip register required in AMNT to store the subtree root. AMNT achieves its design goal of limiting additional hardware components on-chip and in memory. In addition, as discussed in the next section, AMNT's performance is agnostic to other features, such as metadata cache size and memory size.

6.7 Recovery

The recovery process requires both the fetch of counter values from memory and the computation of the hashes of data-independent regions. For example, nodes within a level (i.e., siblings or cousins in a BMT) are data independent, however since a parent node cannot be computed without knowing the value of its children, parents and children have a data dependent relationship in hash recomputation. To relieve this data dependency, the re-computed hash values for a level are written back to memory before the next level can start the hash computation. Seeing as the hash computation is both fast and pipelined, we assume that the recovery time is bound by the memory bandwidth. We note that the split of reads and writes in the recovery workload is a ratio of 8:1 (reads:writes) as eight children are to be fetched in order to compute a parent hash (which will be written back to memory). A single Optane DIMM supports around 4 GB/s of total bandwidth when subjected to this mixed read/write sequential workload [1], of which around half of this bandwidth (2 GB/s) is dedicated to reads. Assuming a six-channel machine [31, 35], this provides a total read bandwidth, at recovery, of 12 GB/s to memory, which is the essential performance bottleneck for recovery. We use this bandwidth to generate the data in Table 4, which shows the time it takes to recover each of the baseline and state-of-the-art configurations after a system failure.

Unlike prior approaches, recovery time in AMNT scales with the level in which the subtree root is placed and is reconfigurable. For example, with the AMNT subtree root configured at level 3 of the BMT, it has a slower recovery time than Anubis [85] (as shown in Figure 4). In the event that a service provider cannot tolerate long periods of downtime, AMNT can be re-configured with a subtree root closer to the leaves. For instance, with the subtree root configured at level 4 for a 2TB memory the recovery time is 0.01 seconds (see Table 4).

7 Related Work

7.1 Secure Memory

Secure memory is an active area of research, with significant improvements to the original secure memory design over the last two decades. The state-of-the-art secure memory solutions combine one-time-pads (OTP) and counter-mode encryption done in hardware [8, 28, 58, 71, 79]. For integrity verification, state-of-the-art include the use of modified Bonsai Merkle Trees (BMTs) for performance, fault tolerance and area optimizations. In Morphable Counters, Saileshwar *et.al.* [60] look to further reduce the area overhead of the integrity tree by introducing novel mechanisms for having a multi-arity tree that can be adapted based on the security and performance needs of the system. Synergy [61] is a mechanism that combines properties of secure memory along with Error Correcting Codes (ECC) in memory to compact the amount of memory overhead required to store the data HMACS. Most of these techniques are orthogonal to the proposed work and can be implemented in conjunctions with AMNT.

7.2 Hardware for SCM

Hardware for storage class memory, and in particular, for enforcing crash-consistency of data, has had two major themes in the literature: memory persistency and persistent transactions. The earlier theme focused on minimal hardware primitives for enforcing ordering of data updates into persistent memory [22, 36, 39, 49], broadly termed memory persistency [51]. The latter theme explored the support of transactional updates to persistent memory within the architecture [15, 37, 68, 77, 84], in the style of hardware transactional memory [30]. Battery-backed caches [62] provide new challenges and opportunities for application persistence [81, 83], and the persistence of security metadata [4, 26, 32], but knowing how much battery is required for data-dependent flushing remains an open issue, and can further complicate hardware overheads.

7.3 Secure SCM

Anubis [85], much like AMNT, provides low run-time overhead and fast recovery. However, Anubis takes a fundamentally different approach to crash consistency, and enforcing this protocol has implications on runtime behaviors and overheads. Anubis tracks the metadata address currently residing in the volatile state (i.e., in the metadata cache) in persistent memory. This region of memory, dubbed a "shadow table", essentially creates a log of all potentially stale metadata values in the persistent state at the time of a crash. Thus, any update to the cache state (misses, evictions, and writebacks) require updating the state of the in-memory shadow table. However, because this table resides in untrusted memory, it too needs to be protected by an auxiliary "shadow Merkle tree" to preserve its integrity. Thus, updates to the shadow table also result in updates to the auxiliary "shadow Merkle tree." While this protocol results in low runtime overhead in most cases, it makes the case of a metadata cache miss more expensive. Anubis works from the observation that updating this table is infrequent because the metadata cache tends to exhibit good locality. We further note in our evaluation that updating the shadow table needs to be atomic with updating the tree state, and there may be multiple shadow table updates on a single authentication (due to multiple misses in the metadata cache). Furthermore, it requires caching the entire shadow Merkle Tree on-chip to avoid even more memory persists per data access. By contrast, the AMNT protocol trades off tracking of the stale nodes for minimal area overheads while still bounding the recovery time. Furthermore, AMNT is not bound by the hit rate of the metadata cache. Instead, it is dependent on a hot region tracking mechanism, in which the complexity is offloaded to software.

Bonsai Merkle Forest (BMF) [26] is a protocol designed to dynamically reduce the leaf-to-root write path for frequently accessed nodes. To do so, BMF extends the persistent register used to store the BMT root into a non-volatile metadata cache to store several roots of frequently accessed values (dubbed the "persistent root set"). BMF determines which BMT nodes qualify to be stored in the persistent root set, and tracks accesses frequency counters for the cached blocks in the non-volatile metadata cache. On a pre-determined interval, BMF uses these access frequency counters to "prune" a frequently accessed root into its most frequently accessed children or "merge" colder roots into their parent node. This mechanism ensures that all nodes in the BMT are covered by a persistent root, which is an important correctness property for this approach. However, this property also implies that it is infeasible to perform a hybrid metadata persistence strategy. As a result, it suffers from the limitations of whichever crash consistency policy it implements. Unlike BMF, AMNT does not assume full leaf coverage of a fast subtree, so it can improve runtime overhead without jeopardizing fast recovery. AMNT's protocol does not entail incremental changes to track hot nodes, and as a result exhibits better performance than BMF. Furthermore, AMNT does not require buffers of non-volatile memories on-chip, to cache the large number of roots required for full leaf coverage as BMF does.

Osiris's [82] method of persistence hinges on reducing memory accesses when updating the tree in favor of cache updates, only persisting integrity tree leaves to memory every *n* updates, where *n* is a set interval. Persist Level Parallelism [25] focuses on fast integrity tree updates and explores the benefits of having parallel updates of the BMT under strict conditions that guarantee correct crash recoverability. However, these works are not *dynamic* to changing application behavior and as a result do not see the benefits in terms of recovery time in the case of Osiris nor normal case runtime overhead in the case of Persist Level Parallelism.

Other works that have proposed using a fast subtree can be classified into two works that leverage indirection [6, 76] and those that use data addresses to determine its membership or non-membership in the fast subtree [56]. AMNT uses data addresses to determine membership in the fast subtree protocol. This design choice is an advantage over approaches that use indirection for two reasons: (1) approaches that leverage indirection cannot begin until some information is fetched that determines which authentication protocol to use, and (2) the logic controlling this indirection leads to significant on-chip area and in-memory storage overheads (additional caches, in-memory queues, etc. [31, 57]). Assure describes a protocol where a single subtree is tracked to reduce the authentication and update path length for frequently accessed data [56]. However, this work does not target multiple persistence strategies. AMNT intentionally addresses this non-trivial question, and results in performance benefits.

Some prior works consider multiple persistence protocols in secure memory to benefit performance [10, 65]. For example, Triad-NVM describes a protocol in which entire levels of the tree conform to a particular persistence protocol [10]. Such a behavior is similar to leaf persistence, where leaves and some number of ancestral levels of the leaves are written-through to main memory. AMNT similarly implements multiple persistence strategies, but determining which strategy to use is done dynamically based on application behavior. OMT describes a protocol for hybrid embedded volatile and non-volatile memories, that leverages a single integrity tree with different strategies for data in different devices [65]. AMNT abstracts well to a hybrid SCM-DRAM machine as it does not require significant protocol or hardware changes. AMNT protects SCM, and a traditional BMT protects DRAM. This solution only requires an additional (volatile) register for the BMT and knowledge at the memory controller of the SCM/DRAM physical address partition. Furthermore, AMNT dynamically leverages leaf metadata persistence in the most frequently accessed SCM region.

8 Conclusion

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

9 Acknowledgment

We thank the anonymous reviewers for their helpful comments. This work was supported by NSF grant #1909715, NSF grant #2239373 (CAREER), and a grant from Google through the exploreCSR program.

A Artifact Appendix

A.1 Abstract

This artifact includes the implementation of AMNT, and other prior art in secure memory in gem5 v22.0.0.2; and the OS modifications for AMNT++ in Linux v4.14.

A.2 Artifact check-list (meta-information)

- **Algorithm:** We propose A Midsummer Night's Tree, a fast subtree tracking mechanism for Secure SCMs. We optimize the protocol by implementing a lightweight OS modification in the physical page allocator called AMNT++.
- **Program:** We use gem5 v22.0.0.2 for simulation. The code modifications can be found in *src/mem/mee/amnt.{hh,cc}*. The directory *src/mem/mee* has the implementation of the baseline protocol and other prior works. The modified OS (AMNT++) can be found in the *linux-build* directory.
- **Compilation:** To build gem5, once inside the main gem5 directory run the following command: `scons build/ARM/gem5.opt -j 'nproc'`. Note `'nproc'` can be replaced with a smaller number of cores to use for compilation. To build the modified Linux kernel (optional), once inside the linux-build directory run the following command where *nproc* is replaced by the number of available cores on the machine: `make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- gem5_defconfig && make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- -j 'nproc'`.
- **How much disk space required (approximately)?:** 72GB.
- **How much time is needed to prepare workflow (approximately)?:** Approximately 1 hour.
- **How much time is needed to complete experiments (approximately)?:** 6-48+ hours per test, depending on the benchmark. For long running tests, they can be constrained to a more reasonable runtime by specifying a consistent *maxinsts* for all configurations in a benchmark.
- **Publicly available?:** Yes.
- **Archived (provide DOI)?:** 10.5281/zenodo.10795547

A.3 Description

A.3.1 How to access. The artifact can be downloaded from https://doi.org/10.5281/zenodo.10795547.

A.3.2 Hardware dependencies. Each experiment requires single core (ISA agnostic) and about 8GB of RAM. Given the long running requirements of each experiment, we suggest running experiments for each benchmark and each protocol in parallel with as many cores as available in the target machine.

A.3.3 Software dependencies. Gem5 works best on Ubuntu, and is largely unsupported on other OSes.

A.4 Installation

Unzip the provided zip files such that the decompressed *dist* and *checkpoints* directories are subdirectories of the unzipped *gem5* directory. Then, compile the simulator from the provided source in the *gem5* directory.

```
$ wget https://zenodo.org/records/10795547/files/gem5.zip
$ unzip gem5.zip
$ wget
↪  https://zenodo.org/records/10795547/files/checkpoints.zip
$ wget https://zenodo.org/records/10795547/files/dist.zip
$ wget
↪  https://zenodo.org/records/10795547/files/main_results.zip
$ wget
↪  https://zenodo.org/records/10795547/files/parse_results.zip

$ unzip checkpoints.zip
$ unzip dist.zip
$ unzip main_results.zip
$ unzip parse_results.zip
$ mv checkpoints gem5/
$ mv dist gem5/
$ mv main_results gem5/
$ mv parse_results.py gem5/

$ sudo apt install build-essential git m4 scons zlib1g
↪  zlib1g-dev libprotobuf-dev protobuf-compiler
↪  libprotoc-dev libgoogle-perftools-dev python3-dev
↪  python3
$ cd gem5
$ scons build/ARM/gem5.opt -j 4
```

Note, compiling gem5 with scons will print a benign warnings concerning libpng and HDF5 C++ libraries.

A.5 Evaluation and expected results

To run the experiments use the provided *run_benchmark.sh* script to run each secure memory protocol for each benchmark. The output from each simulation is stored in the *stats.txt* file associated with the output, and can be cleanly parsed by calling `python3 parse_results.py` to recreate the main result of the paper (Figure 4).

The *run_benchmark.sh* takes in six inputs:

1. The type of operating system to be used: modified (for amnt++) or unmodified.
2. The benchmark suite to run: parsec or parsec_multiprog.
3. The name of the benchmark to be run on the simulator.
4. The type of run to be run: ParsecSP-HW, ParsecSP-HWSW (for amnt++), ParsecMP-HW, or ParsecMP-HWSW (for amnt++).
5. The number of instructions to simulator: suggested, 1 billion.
6. The protocol to simulate : leaf, strict, anubis (prior work), bmf (prior work), amnt (use benchmark type to specify amnt++).

The *run_benchmark.sh* script can be run with the $-h$ option to see the list of arguments that it excepts and an explanation for each. For example, to run *blackscholes*:

```
$ bash run_benchmark.sh unmodified parsec blackscholes
↪  ParsecSP-HW 1000000000 leaf
$ bash run_benchmark.sh unmodified parsec blackscholes
↪  ParsecSP-HW 1000000000 strict
$ bash run_benchmark.sh unmodified parsec blackscholes
↪  ParsecSP-HW 1000000000 anubis
$ bash run_benchmark.sh unmodified parsec blackscholes
↪  ParsecSP-HW 1000000000 bmf
$ bash run_benchmark.sh unmodified parsec blackscholes
↪  ParsecSP-HW 1000000000 amnt
$ bash run_benchmark.sh modified parsec blackscholes
↪  ParsecSP-HWSW 1000000000 amnt
```

Running this for each benchmark produces all of the output required to recreate the results in Figure 4 of the original manuscript.

References

[1] 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.

[2] Linux kernel documentation. https://www.kernel.org/doc/html/v4.9/kernel-documentation.html. Accessed: 2022-07-06.

[3] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[4] Mohammad Alshboul, Prakash Ramrakhyani, William Wang, James Tuck, and Yan Solihin. Bbb: Simplifying persistent programming using battery-backed buffers. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 111–124. IEEE, 2021.

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

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

[7] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing. In *Proc. International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[8] Ittai Anati, Frank Mckeen, Shay Gueron, Huang Haitao, Simon Johnson, Rebekah Leslie-Hurd, Harish Patil, Carlos Rozas, and Hisham Shafi. Intel software guard extensions (Intel SGX). In *Tutorial at International Symposium on Computer Architecture (ISCA)*, 2015.

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

[10] Amro Awad, Mao Ye, Yan Solihin, Laurent Njilla, and Kazi Abu Zubair. Triad-nvm: Persistency for integrity-protected and encrypted non-volatile memories. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 104–115, 2019.

[11] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008.

[12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[13] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 41–42, 2018.

[14] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):1–33, 2017.

[15] Miao Cai, Chance C. Coats, and Jian Huang. Hoop: Efficient hardware-assisted out-of-place update for non-volatile memory. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 584–596, 2020.

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

[17] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applicat ions*, OOPSLA '14, pages 433–452, New York, NY, USA, 2014. ACM.

[18] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX security symposium*, volume 5, page 146, 2005.

[19] Zhengguo Chen, Youtao Zhang, and Nong Xiao. Cachetree: Reducing integrity verification overhead of secure nonvolatile memories. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(7):1340–1353, 2020.

[20] Siddhartha Chhabra, Brian Rogers, Yan Solihin, and Milos Prvulovic. Secureme: a hardware-software approach to full system security. In *Proceedings of the international conference on Supercomputing*, pages 108–119, 2011.

[21] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 105–118, New York, NY, USA, 2011. ACM.

[22] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 133–146, New York, NY, USA, 2009. Association for Computing Machinery.

[23] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy*, pages 763–780. IEEE, 2015.

[24] Fabian Fleischer, Marcel Busch, and Phillip Kuhrt. Memory corruption attacks within android tees: a case study based on op-tee. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, pages 1–9, 2020.

[25] Alexander Freij, Shougang Yuan, Huiyang Zhou, and Yan Solihin. Persist level parallelism: Streamlining integrity tree updates for secure persistent memory. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 14–27. IEEE, 2020.

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

[27] Blaise Gassend, G Edward Suh, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. Caches and hash trees for efficient memory integrity verification. In *Proc. International Symposium on High Performance Computer Architecture (HPCA)*, 2003.

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

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

[30] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, page 289–300, New York, NY, USA, 1993. Association for Computing Machinery.

[31] Takahiro Hirofuchi and Ryousei Takano. A prompt report on the performance of intel optane dc persistent memory module. *IEICE TRANSACTIONS on Information and Systems*, 103(5):1168–1172, 2020.

[32] Jianming Huang and Yu Hua. Ensuring data confidentiality in eadr-based nvm systems. *IEEE Computer Architecture Letters*, 21(2):153–156, 2022.

[33] Yutaka Ito and Yuan He. Apparatus and methods for refreshing memory, 2019. U.S. Patent 11062754B2.

[34] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and O perating Systems*, ASPLOS '16, pages 427–442, New York, NY, USA, 2016. ACM.

[35] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[36] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Efficient persist barriers for multicores. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 660–671, New York, NY, USA, 2015. Association for Computing Machinery.

[37] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. Atom: Atomic durability in non-volatile memory through hardware logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 361–372, 2017.

[38] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B Lee. Nohype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th annual international symposium on Computer architecture*, pages 350–361, 2010.

[39] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. Delegated persist ordering. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

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

[41] Tamara Silbergleit Lehman, Andrew Douglas Hilton, and Benjamin C. Lee. MAPS: Understanding metadata access patterns in secure memory. In *Proc. International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2018.

[42] Mengya Lei, Fan Li, Fang Wang, Dan Feng, Xiaomin Zou, and Renzhi Xiao. Secnvm: An efficient and write-friendly metadata crash consistency scheme for secure nvm. *ACM Transactions on Architecture and Code Optimization (TACO)*, 19(1):1–26, 2021.

[43] David Lie, Thekkath Chandramohan, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *SIGPLAN Notices*, 2000.

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

[45] Haocong Luo, Ataberk Olgun, Abdullah Giray Yağlıkçı, Yahya Can Tuğrul, Steve Rhyner, Meryem Banu Cavlak, Joël Lindegger, Mohammad Sadrosadati, and Onur Mutlu. Rowpress: Amplifying read disturbance in modern dram chips. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–18, 2023.

[46] Stephen McLaughlin, Charalambos Konstantinou, Xueyang Wang, Lucas Davi, Ahmad-Reza Sadeghi, Michail Maniatakos, and Ramesh Karri. The cybersecurity landscape in industrial control systems. *Proceedings of the IEEE*, 104(5):1039–1057, 2016.

[47] Ralph C Merkle. Protocols for public key cryptosystems. In *Proc. Symposium on Security and Privacy (SP)*, 1980.

[48] Onur Mutlu and Jeremie S Kim. Rowhammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(8):1555–1571, 2019.

[49] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 135–148, New York, NY, USA, 2017. ACM.

[50] Ismail Oukid, Wolfgang Lehner, Thomas Kissinger, Thomas Willhalm, and Peter Bumbulis. Instant recovery for main memory databases. In *CIDR*, 2015.

[51] Steven Pelley, Peter M Chen, and Thomas F Wenisch. Memory persistency. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 265–276. IEEE, 2014.

[52] Marcus Pendleton, Richard Garcia-Lebron, Jin-Hee Cho, and Shouhuai Xu. A survey on systems security metrics. *ACM Computing Surveys (CSUR)*, 49(4):1–35, 2016.

[53] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. *ACM SIGMETRICS Performance Evaluation Review*, 31(1):318–319, 2003.

[54] pmem.io. Persistent Memory Development Kit, 2017. http://pmem.io/pmdk.

[55] Adnan Siraj Rakin, Md Hafizul Islam Chowdhuryy, Fan Yao, and Deliang Fan. Deepsteal: Advanced model extractions leveraging efficient weight stealing in memories. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1157–1174. IEEE, 2022.

[56] Joydeep Rakshit and Kartik Mohanram. Assure: Authentication scheme for secure energy efficient non-volatile memories. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.

[57] Luiz E Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *Proceedings of the international conference on Supercomputing*, pages 85–95, 2011.

[58] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. Using address independent seed encryption and Bonsai Merkle trees to make secure processors OS- and performance-friendly. In *Proc. International Symposium on Microarchitecture (MICRO)*, 2007.

[59] Brian Rogers, Chenyu Yan, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. Single-level integrity and confidentiality protection for distributed shared memory multiprocessors. In *Proc. International Symposium on High Performance Computer Architecture (HPCA)*, 2008.

[60] Gururaj Saileshwar, Prashant J Nair, Prakash Ramrakhyani, Wendy Elsasser, Jose A Joao, and Moinuddin K Qureshi. Morphable counters: Enabling compact integrity trees for low-overhead secure memories. In *Proc. International Symposium on Microarchitecture (MICRO)*, 2018.

[61] Gururaj Saileshwar, Prashant J Nair, Prakash Ramrakhyani, Wendy Elsasser, and Moinuddin K Qureshi. Synergy: Rethinking secure-memory design for error-correcting memories. In *Proc. International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

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

[63] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762. IEEE, 2015.

[64] Daniele Sgandurra and Emil Lupu. Evolution of attacks, threat models, and solutions for virtualized systems. *ACM Computing Surveys (CSUR)*, 48(3):1–38, 2016.

[65] Rakin Muhammad Shadab, Yu Zou, Sanjay Gandham, and Mingjie 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)*, pages 191–202. IEEE, 2023.

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

[67] Weidong Shi, Hsien-Hsin S Lee, Mrinmoy Ghosh, and Chenghuai Lu. Architecture support for high speed protection of memory integrity and confidentiality in multiprocessor systems. In *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2004.

[68] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. Proteus: A flexible and fast software supported hardware logging approach for nvm. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 178–190, New York, NY, USA, 2017. Association for Computing Machinery.

[69] Sergei Skorobogatov. Data remanence in flash memory devices. In *Cryptographic Hardware and Embedded Systems–CHES 2005: 7th International Workshop, Edinburgh, UK, August 29–September 1, 2005. Proceedings 7*, pages 339–353. Springer, 2005.

[70] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Efficient memory integrity verification and encryption for secure processors. In *Proc. International Symposium on Microarchitecture (MICRO)*, 2003.

[71] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proc. International Conference on Supercomputing (ICS)*, 2003.

[72] Meysam Taassori, Rajeev Balasubramonian, Siddhartha Chhabra, Alaa R Alameldeen, Manjula Peddireddy, Rajat Agarwal, and Ryan Stutsman. Compact leakage-free support for integrity and reliability. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 735–748. IEEE, 2020.

[73] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. Vault: Reducing paging overheads in sgx with efficient integrity verification structures. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 665–678, 2018.

[74] Kenzo Van Craeynest and Lieven Eeckhout. The multi-program performance model: debunking current practice in multi-core simulation. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 26–37. IEEE, 2011.

[75] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[76] Bolin Wang. *Bo-tree: a dynamic merkle tree for enabling scalable memories*. PhD thesis, University of British Columbia, 2022.

[77] Xueliang Wei, Dan Feng, Wei Tong, Jingning Liu, and Liuqing Ye. Morlog: Morphable hardware logging for atomic persistence in non-volatile main memory. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 610–623, 2020.

[78] Yubin Xia, Yutao Liu, and Haibo Chen. Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks. In *Proc. International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[79] Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. Improving cost, performance, and security of memory encryption and authentication. *Proc. International Symposium on Computer Architecture (ISCA)*, 2006.

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

[81] Chongnan Ye, Meng Chen, Qisheng Jiang, and Chundong Wang. Enabling atomic durability for persistent memory with transiently persistent cpu cache. *arXiv preprint arXiv:2210.17377*, 2022.

[82] Mao Ye, Clayton Hughes, and Amro Awad. Osiris: A low-cost mechanism to enable restoration of secure non-volatile memories. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2018.

[83] Bowen Zhang, Shengan Zheng, Zhenlin Qi, and Linpeng Huang. Nbtree: a lock-free pm-friendly persistent b+-tree for eadr-enabled pm systems. *Proceedings of the VLDB Endowment*, 15(6):1187–1200, 2022.

[84] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 421–432, New York, NY, USA, 2013. ACM.

[85] Kazi Abu Zubair and Amro Awad. Anubis: ultra-low overhead and recovery time for secure non-volatile memories. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 157–168, 2019.