ProMT: Optimizing Integrity Tree Updates for Write-Intensive Pages in Secure NVMs

Mazen Alwadi
mazen.alwadi@knights.ucf.edu
University of Central Florida
Orlando, Florida, USA

David Mohaisen
david.mohaisen@ucf.edu
University of Central Florida
Orlando, Florida, USA

Amro Awad
ajawad@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

ABSTRACT

Current computer systems are vulnerable to a wide range of attacks caused by the proliferation of accelerators, and the fact that current system comprise multiple SoCs provided from different vendors. Thus, major processor vendors are moving towards limiting the trust boundary to the processor chip only as in Intel's SGX, and AMD's SME. This secure boundary limitation requires protecting the memory content against data remanence attacks, which were performed against DRAM in the form of cold-boot attack and are more successful against NVM due to NVM's data persistency feature. However, implementing secure memory features, such as memory encryption and integrity verification has a non-trivial performance overhead, and can significantly reduce the emerging NVM's expected lifetime. Previous work looked at reducing the overheads of the secure memory implementation by packing more counters into a cache line, increasing the cacheability of security metadata, slightly reducing the size of the integrity tree, or using the ECC chip to store the MAC values. However, the root update process is barely studied, which requires a sequential update of the MAC values in all the integrity tree levels.

In this paper, we propose ProMT, a novel memory controller design that ensures a persistently secure system with minimal overheads. ProMT protects the data confidentiality and ensures the data integrity with minimal overheads. ProMT reduces the performance overhead of secure memory implementation to 11.7%, extends the NVM's life time by 3.59x, and enables the system recovery in a fraction of a second.

CCS CONCEPTS

• **Security and privacy → Hardware-based security protocols**.

KEYWORDS

Non-Volatile Memory, Secure Memory, Integrity Verification, Integrity Tree.

ACM Reference Format:

Mazen Alwadi, David Mohaisen, and Amro Awad. 2021. ProMT: Optimizing Integrity Tree Updates for Write-Intensive Pages in Secure NVMs. In *2021 International Conference on Supercomputing (ICS '21), June 14–17, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3447818.3460377

ICS '21, June 14–17, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8335-6/21/06…$15.00
https://doi.org/10.1145/3447818.3460377

1 INTRODUCTION

Current computing systems suffer from a very wide attack surface, mainly due to the fact that such systems comprise of tens to hundreds of sub-systems that could be manufactured by different vendors. Vulnerabilities, backdoors, and potentially hardware trojans injected anywhere in the system form a serious risk for confidentiality and integrity of data in computing systems. Rightfully, processor vendors minimize the trust boundaries to only include the processor chips, e.g., AMD's Secure Memory Encryption (SME)[1], Intel's Software Guard Extension (SGX)[19], and Total Memory Encryption (TME)[4]. This trust boundary limitation mandates providing security measures, (i.e., memory encryption and integrity verification), to protect the memory content. Counter mode encryption and integrity trees are used in state-of-the-art schemes to meet these security requirements [18, 20, 42, 43, 45, 47].

Counter mode encryption protects the data confidentiality by assigning an encryption counter for each data cache line, which is used to encrypt/decrypt the data cache line whenever it is written or read to or from the memory [33, 55, 58]. The integrity tree hashes the memory content into a tree structure and keeps the root of the tree in the processor. The root is used to verify the integrity of the data whenever it is read from the memory [19, 22, 39]. To be able to verify the data using the root, it must reflect the most recent state of the memory, which requires updating the whole integrity tree branch for each memory write [11, 34, 58]. Clearly, these security measures are incompatible with the emerging Non-Volatile Memories (NVMs).

Emerging NVMs have a compelling set of features such as data persistency, capacities that are up to 3TBs/socket, and ultra-low idle power consumption [3, 5–8]. On the other hand, NVMs have a limited write endurance and slow power-consuming writes [13, 28, 29, 55]. Due to these features, implementing memory encryption and integrity verification can lead to several problems. First, as the NVMs retain the data during power loss episodes, the used encryption counters and integrity tree nodes need to be persisted atomically with the data to ensure the system's ability to recover, which leads to the second problem. Second, persisting the security metadata can lead to tens of writes for each data write in practical size NVMs. Third, before persisting the updates, the integrity tree root needs to be updated by hashing all the updated branch levels. Clearly, updating the integrity tree root resides on the critical path of the write operation, and for a 3TBs memory, the integrity tree can reach 11 levels and a root, which takes 440-880 cycles to complete using a 40-80 cycle hashing function [20, 31, 44]. Moreover, these

11 updates of the integrity tree, the encryption counter update, the MAC value, and the data cache line need to be held in the write buffer until the root has been successfully updated, thus filling the processor buffers and stalling the execution.

Solutions and Shortcomings. Prior work addressed the first two problems but not the third. For instance, Liu et al. [34], Osiris [55], and SuperMem [60] addressed the crash consistency problem by proposing solutions to recover or persist the encryption counters, but ignored the required time to rebuild the integrity tree. Anubis [58], Phoenix [11], Triad-NVM [13], and SheildNVM [53] proposed schemes to rebuild the integrity tree after crashes. However, the overheads of updating the integrity tree root was rarely studied until recently. Freij et al. [20] discussed the overheads of crash consistency requirements in light of the enforced persist ordering, and proposed a Persist Level Parallelism (PLP) optimizations to reduce these overheads. However, the PLP optimizations are discussing the ordering, possible overlapping of the integrity tree updates, and assume a pipeline large enough to perform the updates of the WPQ entries, which can be 32-64 entries. Clearly, assuming this number of hashing engines is not realistic, as these engines have to be located in the secure region—inside the processor chip. Additionally, the proposed scheme works on updating the integrity tree root, then persisting the data, MAC, and encryption counter but not the integrity tree intermediate nodes. While the followed technique is sufficient to recover the system, rebuilding the integrity tree can take hours for practical size NVMs [11, 58].

Our Approach. To bridge this gap and address the aforementioned problem, we propose ProMT. In ProMT, we make the observation that not all memory pages are updated at the same frequency. Moreover, to ensure the system's data integrity and timely updates of the integrity tree root, updating the integrity tree protecting the whole memory is not necessary. Thus, ProMT shortens the root's update path by using a separate small integrity tree, which is used to protect the hot pages. To do so, ProMT tracks the hotness of memory pages using descriptor blocks that are arranged in a Multi-Queue (MQ) structure, cached, and controlled by the memory controller. These hot pages are dynamically detected and assigned to the hot tree, which effectively reduces the time required to update the root by the difference in the number of levels between the two integrity trees. Additionally, ProMT allows the system to recover from crashes in a fraction of a second, and effectively reduces the number of writes required to enable the system's recovery. Moreover, ProMT avoids the re-encryption overheads that would be required to map the pages to a hot region. To evaluate our scheme, we use the Gem5 [16] simulator to run 10 memory intensive applications from SPEC2006 benchmark suite [23], and 7 persistent multi-threaded applications that were developed in-house, similar to [10, 35]. Compared to an eagerly updated integrity tree, ProMT improves the performance by 63.6%, reduces the writes by 3.59x, and enables the system's recoverability in 3ms, compared to a lazy-update scheme.

In summary, we make the following novel contributions:

- We propose ProMT, a novel memory controller design that significantly reduces the overhead of updating the integrity tree root, reduces the write traffic to the memory, and enables the system recovery in a fraction of a second.
- We discuss several design options of ProMT, and provide a detailed analysis about how these options can impact the system's performance.
- We discuss how ProMT can be integrated with state-of-the-art encryption, integrity verification, and secure memory schemes.

The rest of the paper is organized as follows. In section 2 we discuss the related background concepts and motivate our work. In section 3, we discuss our design, potential design options, and the impact of our design on the overall system security. In section 4, we discuss the used applications and the testing methods we followed for evaluating our work. In section 5, we analyze the evaluation results and the performance of our scheme, followed by the related work and conclusion in sections 6 and 7.

2 BACKGROUND AND MOTIVATION

In this section, we introduce the background and motivation. First, we review the threat model followed by various related concepts that will help the reader understand our work, then make the case for our contributions by a concrete motivation.

2.1 Background

Threat Model. Similar to state-of-the-art schemes in secure-memory implementation [11–15, 46, 55, 56, 58, 60], we assume a passive attacker capable of scanning the memory and snooping the memory bus. We also assume an active attacker capable of dropping packets, tampering with the memory content, and replying old memory data. Thus, our scheme limits the trust boundaries to the processor chip only. Finally, memory access pattern leakage [48, 49], timing side-channel leakage [49], and power analysis [36] attacks are all beyond our threat model and the scope of this paper.

Emerging Non-Volatile Memories (NVMs). Emerging NVMs blur the boundary between storage and memory systems by introducing a set of compelling features . Such NVMs feature access latencies comparable to DRAM, byte addressability, data persistency, ultra-low idle power consumption, and high density [11–15, 55, 56, 58, 60]. Due to these features, researchers are expecting NVMs to be integrated into memory systems as a main memory [30, 32, 58, 60], as part of the main memory as in hybrid systems [37, 38, 50, 51], or as extensions to the main memory as in fabric-attached memory systems [25–27]. However, such NVMs suffer a limited write endurance and power-consuming writes.

While NVMs data persistency might be the most attractive feature, as it enables hosting persistent data such as filesystems and checkpointing. Data persistency facilitates data remanence attacks, which requires security measures to ensure the NVMs' data confidentiality and integrity.

Counter Mode Encryption. The counter mode encryption is used in state-of-the-art secure memory schemes [11–15, 42, 43, 46, 47, 55, 56, 58, 60] to protect the data confidentiality. Figure 1 shows the split counter mode encryption and how it works. The split counter mode arranges one major counter and 64 minor counters in a single cache line. The major counter has a size of 64-bits and the minor counters have a size of 7-bits each [11, 58]. To complete the encryption/decryption, an initialization vector (IV) is formed using the major counter, minor counter, the data page offset, the data page

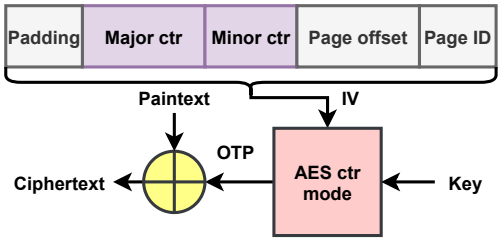

Figure 1: Split counter-mode encryption.

ID and some padding. The IV is encrypted using an AES engine with a secure processor key to generate a One-Time Pad (OTP). The OTP is then xored with the plaintext/ciphertext to complete the encryption/decryption.

The counter mode is widely used due to its performance and security advantages. In terms of performance, the counter mode encryption allows overlapping the OTP generation time with the memory read, which leaves only the xor operation exposed to the critical path of the memory read. In terms of security, the counter mode thwarts dictionary attacks, snooping attacks, and known-plaintext attacks [11, 33, 55, 58]. However, in order to ensure the security of the counter mode encryption, encryption counters reuse is prohibited, as it facilitates known-plaintext attacks. Note that the counter mode encryption ensures the temporal and spatial uniqueness of the IV, as it integrates the encrypted cache line address in the IV, and increments the associated minor counter each time the cache line is written back to the memory. Whenever a minor counter overflows, the page's major counter is incremented, all the minor counters in the same cache line are reset, and the page is re-encrypted using the new counters. Despite the advantages of counter mode encryption, it does not ensure memory integrity, which is achieved using an integrity tree.

Integrity Trees. An integrity tree is a tree of hashes, in its most basic form, where the data cache lines are hashed together to form the first level of the tree, then each N nodes of the first level are hashed together to generate the second level and so on. The same process continues until a single node is generated, which is referred to as the root. The number of nodes hashed together to generate the next level is typically referred to as the tree arity. The root is always kept in the secure region and used to verify the integrity of the memory content [11, 13, 19, 22, 39, 58]. Rogers et al. [39] proposed the Bonsai-Merkle Tree (BMT), which builds the Merkle tree over the encryption counters, and protects the data cache lines with MAC values calculated over the data and the encryption counters. Integrity trees were developed into two major types, the general Merkle Tree (MT), and the Tree of Counters (ToC) used in Intel's Software Guard Extension (SGX) [19, 22, 47]. Each of these forms has a different arity, update style, and node structure as we discuss below.

General Merkle Tree. This tree is shown in Figure 2-A, and protects the encryption counters by hashing the encryption counters and forming the tree. The root of the tree is kept in the processor and used to verify the integrity of the encryption counters. Whenever an encryption counter is read from the memory, the whole MT branch is rehashed to calculate a root value. The calculated root value is then compared against the processor stored root, and if they match, the encryption counter's integrity is verified. A faster way to verify the integrity can be achieved by stopping the verification process with the first MT node cache hit, as cached nodes' integrity were verified when they were fetched. While the general MT is used to verify the encryption counters' integrity, data integrity is protected using Keyed Message Authentication Codes (HMAC) values [39, 40, 43, 52], as shown in Figure 2-C. The MAC values are calculated over the encrypted data, encryption counter, the processor key, and the data address as:

$$\text{MAC} = H_k(\text{encrypted data}, \text{EC}, \text{address}) \quad (1)$$

where H_k is the keyed MAC function. Including the data and EC prevents replaying old pairs of {data, MAC}, using the processor key prevents attackers from forging {data, MAC} pairs, using the encryption counter ensures the temporal uniqueness, and the address is used to ensure spatial uniqueness, which prevents splicing attacks [22, 39].

The general Merkle Tree has three main differences from the ToC. First, the general Merkle Tree does not associate any MAC values with the encryption counter block, which allows higher arity. Second, each node in higher levels is just a hash of its direct children [11, 58]. Thus, the update process of the general MT is a serial process, as the update of any level cannot start until the update of its direct child level finished updating [11, 58]. Third, the general MT can be rebuilt if the encryption counters were persisted. However, rebuilding the tree can take several hours for practical size NVMs as discussed by Anubis [58].

Tree of Counters. As shown in Figure 2-B, the ToC serves the same purpose by protecting the encryption counters' integrity. However, the ToC has some differences in terms of encryption counters' arrangement and intermediate tree nodes.

The counter node encryption in ToC has 8 encryption counters of 56-bit each, a 56-bit MAC value, and 8 unused bits. The MAC value is calculated over the node's encryption counters, and a version from the parent node. While the lowest level counters are used to encrypt the data, the intermediate node's counters are typically referred to as versions, and are not associated with the data. Whenever a data cache line is written back to the memory, the associated encryption counter is incremented, which leads to incrementing the version in the encryption counter's parent, and updating the encryption counter's MAC [11, 19, 22, 43, 46, 47, 58]. Note that the update is propagated until the root is updated. Since nodes' updates do not need to wait for child nodes to finish updating, the ToC update process can be parallelized [19, 22, 58].

Write Atomicity. Write atomicity is required to enable persistent security of the secure NVM, as if a crash happened and the security metadata were not up-to-date, the security metadata will fail to verify the memory's integrity [34, 55, 56], which can lead to either losing terabytes of data or accepting the risk that data might be tampered with.

To avoid the inconsistency of data and its associated security metadata, the existing memory controller's persistent buffer known as the Write Pending Queue (WPQ) is used [2, 13, 41]. The WPQ is provided with enough power by the Asynchronous DRAM Refresh (ADR) feature to flush its content to the NVM in case of crashes.

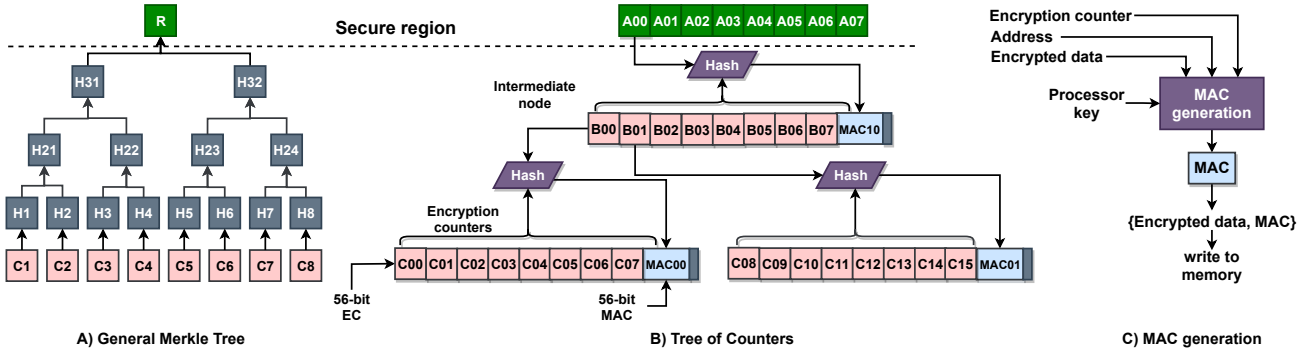

Figure 2: Integrity trees and MAC generation.

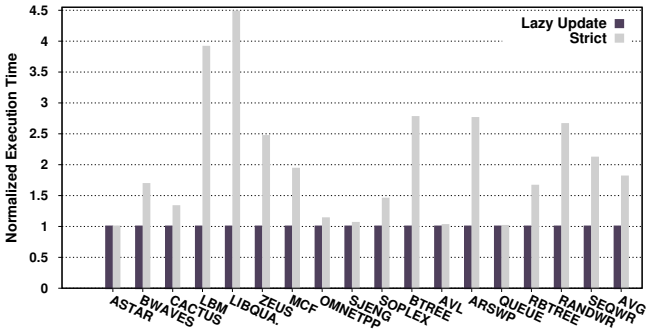

Figure 3: Performance overhead of Merkle Tree root updates compared to a write-back scheme.

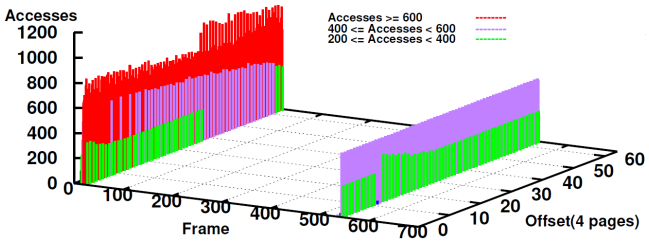

Figure 4: Memory accesses of RBTREE application.

2.2 Motivation

As discussed earlier, updating the integrity tree root is a very costly operation. The performance overhead of this operation stems from the sequential hashing process of the whole MT branch. Moreover, the latency of each hashing operation can be 40-80 cycles, which resides on the critical path of the memory writes [20, 31, 44]. Moreover, the MT branch depth exceeds 10 levels for practical size NVMs, and due to the atomicity requirement, these updates need to be held in the write buffer until the root is updated. Figure 3 shows the performance overhead caused by the root updates, compared to an encrypted memory that uses a write back policy and does not provide the crash consistency.

As shown in Figure 3, the performance overhead of updating the MT root can reach to 1.8x on average. The performance overhead is directly related to the number of writes the application is sending to the memory and the number of the MT levels. As we have no control over the number of writes the application is making, we aim to reduce this overhead by reducing the number of MT levels. As the memory pages are not accessed at the same rate, protecting the most updated pages with a small MT can reduce the overhead. Figure 4 shows the spatial distribution of the memory accesses.

In Figure 4, the y-axis represents the offset, which is a group of 4 pages. The x-axis shows the frame number, which contains 64 groups. The z-axis shows the number of accesses to each group. Clearly, protecting the hot pages with a smaller Merkle Tree can reduce the performance overhead. However, protecting all these hot pages will require a decently sized Merkle Tree, which will not yield much of improvement. On the other hand, the figure shows the spatial distribution of the accesses, combining this with the temporal hotness of each page can allow us to have a small Merkle Tree, which can improve the performance significantly.

3 DESIGN

ProMT aims to improve the performance of integrity protected memories by reducing the overhead caused by the frequent integrity tree root updates. The main reason of these overheads is the number of MT nodes to be updated for each write, which can be reduced if the number of MT levels is reduced. ProMT relies on the fact that not all memory pages are updated at the same rate. Therefore, ProMT uses an additional small MT to protect the hot pages and reduce the number of levels to be updated for each write.

3.1 Design overview

Reducing the overheads of the MT can be achieved by reducing the number of its levels, which reduces the number of MAC calculations required to update the MT root. As discussed in section 2.2, we can exploit the observation that memory pages are updated at different rates. Thus, we can reduce the MT depth by building a small MT over the hot pages.

Applying such a technique requires careful consideration as it can cause higher overheads due to three main challenges. First, integrity trees are rigid structures used to protect a consecutive memory region. Additionally, the application's data can be scattered across the whole physical address space, which makes using a

dynamically growing integrity tree very complicated, if possible in the first place. Second, choosing the hot region to protect with the hot MT. Using a second integrity tree requires building the tree over a different set of encryption counters, which requires re-encrypting the pages as they are mapped/evicted to/from the hot region. Third, detecting the hot pages dynamically, with high accuracy, and low overheads. Identifying the hot pages requires tracking the accesses to all memory pages, which can cause a high performance overhead and requires non-realistic on-chip storage.

We discuss the potential design options below, followed by our chosen design.

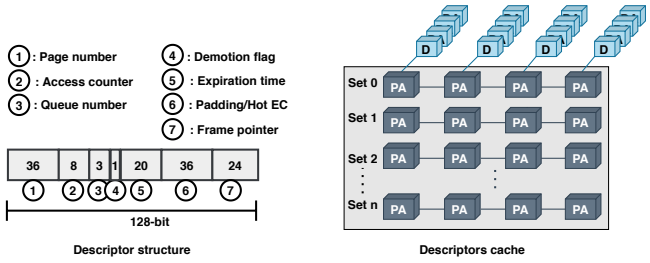

Figure 5: Arrangement and caching of descriptors.

3.2 Design options

The first design option is to allocate a dedicated hot memory region to which we copy the hot pages. The hot region should have a separate set of encryption counters and a separate integrity tree. However, this option requires copying the page once declared hot, then copy it back and re-encrypt it once evicted. Such a scheme, in most cases, will incur more writes, MAC calculations, and higher performance overhead.

The second option, is to use a different set of encryption counters and a separate integrity tree but without allocating a dedicated memory region. Once a hot page is detected, the encryption counters of the hot MT are used to re-encrypt the page. This scheme does not require copying the data pages but requires re-encryption for each insertion/eviction.

The third option is similar to the second option in terms of counters, integrity tree, and region allocation. However, in this option, instead of re-encrypting the page at eviction time, the global MT's encryption counter value is replaced with the hot encryption counter's value and the integrity tree is updated. While this technique can eliminate half of the re-encryption writes, this is only possible if the hot encryption counter value is larger than the global MT's one, which is necessary to preserve the security of counter mode encryption. Moreover, this scheme will increase the rate of incrementing the encryption counters, as the hot region encryption counters will resume using the hot counter value used for the previous page. Thus, updating the encryption counters for the pages protected using the same encryption counter of the hot MT at almost the sum of all updates for those pages, which leads to a faster overflow of the counters.

As discussed above, the previously mentioned design options are costly in terms of performance, MAC calculations, and NVM writes. While the last design option can partially solve the problem by eliminating half of the writes, we observe that it can completely eliminate the writes if the page's hot encryption counter has the same value as the page's global MT encryption counter, which leads us to ProMT design.

3.3 ProMT design

ProMT design is built on the observation that if the page's associated encryption counters in the global MT and the hot MT are equal on eviction and insertion, there will be no need to re-encrypt the page and it would be sufficient to just update the upper MT levels to ensure the correctness of integrity verification. Thus, ProMT does not use a different set of encryption counters for the hot MT, but remaps the hot page's encryption counter from the global MT to the hot MT. Thus, the encryption counter updates of the hot page are no longer propagated to the global MT, and the updates are propagated to the hot MT instead. On eviction, it is sufficient to propagate the update to the global MT once. Assuming L_{G-M_T} is the number of levels in the global MT, and L_{H-M_T} is the number of levels in the hot MT, and k is the number of writes to a hot page P. The number of avoided MAC calculations when the page is protected by the hot MT is:

$$\text{MACs Reduction} = (L_{G-M_T} - L_{H-M_T}) * (k-1) \quad (2)$$

As shown in Equation 2, the number of MACs required to update the integrity tree protecting a hot page can be reduced to $\approx L_{H-M_T}/L_{G-M_T}$. Note that ProMT design does not allocate a dedicated memory region, which eliminates the need for copying the pages from/to the hot region. Additionally, ProMT uses the same encryption counter used by the global MT, which eliminates the need for the re-encryption. Finally, ProMT allows assigning any page on the physical address space to the hot MT without the need to expand or dynamically grow the hot MT. However, to enable this dynamic remapping of the pages, tracking the encryption counters of the hot MT becomes a crash consistency requirement as we discuss below.

3.3.1 Hot pages detection. Identifying hot pages is done by tracking the number of writes to each page. For that, ProMT creates a descriptor block for each accessed page and organizes these blocks in a Multi-Queue (MQ) structure, which was originally introduced to rank disk blocks and used later by Ramos et al. [38] for page placement in hybrid memory systems. Each descriptor block contains the page number, an access counter to count the number of accesses, a queue number to identify the queue to which the block belongs, an expiration time used for evictions from the hot region, a frame pointer to the next block in the queue, and the encryption counter number in the hot MT if the page is hot.

3.3.2 Page tracking. A descriptor block is created and inserted to Q0 on the first access to the page. A descriptor block is promoted to Q1 when the access counter reaches 2 accesses, and promoted to Q2 on 4 accesses and so on. Whenever a page gets an access the descriptor's expiration time is set to the *current time + life time*, if a descriptor expiration time is reached, the descriptor gets demoted, its expiration time gets reset, then inserted to the tail of the below queue. If the descriptor gets demoted two consecutive times, it gets inserted to the tail of Q3 which we use to select victims for eviction. To reduce the overhead of checking the MQ, the demotions are

performed at the end of each epoch by checking only the head of each queue, as insertion, promotion, and demotions are inserted to the tail of each queue, the head of each queue represents the least recently accessed block. For more details about the MQ, we refer the readers to Ramos et al. [38].

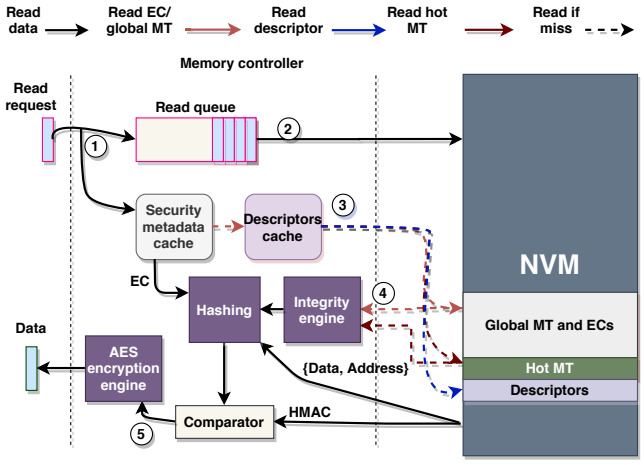

Figure 6: ProMT read operation.

Optimization. Since the MQ size can reach a size that is not feasible to keep in the processor chip, we store the MQ in the NVM and use a small cache in the memory controller to cache the MQ descriptors. To ensure retrieving the descriptor block in a single memory access and increase the descriptors cacheability, we group each four descriptor blocks in a single cache line and store them in a table. The table entry number is calculated using a modules operation of the page physical address and the number of groups. We directly map the pages into the table entries to prevent collisions and enforce a single access retrieval. The descriptor block and the descriptors cache are shown in Figure 5.

3.3.3 ProMT's read operation. As shown in Figure 6, upon receiving a read request, the memory controller starts by forwarding the request to the NVM, and requests the associated encryption counter if not present in the security metadata cache. Then, checks if the requested cache line is protected by the global or the hot MT by checking the descriptors cache, then requests the required global/hot MT nodes from the security metadata cache. After receiving the data cache line and the associated security metadata, the memory controller verifies the encryption counter's integrity using the global/hot MT, then verifies the integrity of the cache line data using the HMAC that is calculated over the data and its encryption counter.

3.3.4 ProMT's write operation. As shown in Figure 7, upon receiving a write request, the memory controller starts by checking the page hotness and increments the access counter in its descriptor block. If the page reaches the hotness threshold, a victim page is selected for eviction while the data is being encrypted. Then, the data is written back, the HMAC is updated, and the integrity updates are reflected to the hot MT, and the evicted page's integrity updates are reflected to the global MT. If the page did not reach the hotness

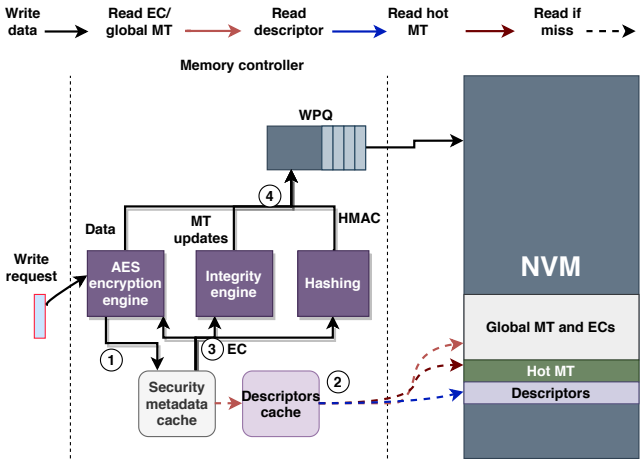

Figure 7: ProMT write operation.

threshold, the integrity updates are reflected to the global MT. Note that when a page is inserted to the hot MT, the total number of writes to update both integrity trees is $L_{H-M_T} + L_{G-M_T}$.

3.4 ProMT's without persistence

Since ProMT uses a general MT to protect the hot region pages, updates to the hot MT are not essential for the crash consistency as long as the root of the hot MT is up-to-date. Due to the small size of the hot MT, and the ability to regenerate the general MT, ProMT can be optimized to reduce the number of writes to the NVM. ProMT No Persist (ProMT-NP) reduces the number of writes at the cost of extra recovery time, which is required to regenerate the hot MT during the recovery phase.

3.4.1 ProMT's impact on recoverability. Since the global MT and the hot MT are eagerly updated, the security metadata is ensured to be consistent with the data. However, as the encryption counters can be protected by either of the integrity trees, we need to be able to identify the encryption counters that were protected by the hot MT before the crash. To identify these encryption counters, we persist the addresses of the hot MT encryption counters.

ProMT-NP relaxes persisting the updates of the hot MT, and relies on rebuilding the MT during the recovery phase to ensure crash consistency. Thus, whenever a page is mapped to the hot MT, the page's encryption counter's address needs to be persisted to the tracking region. In the recovery phase, the memory controller iterates over the tracking region and rebuilds the hot MT, then the system is recovered.

3.5 Design discussion

3.5.1 ProMT's hardware modifications. ProMT modifies the memory controller to check if the page is mapped to the hot MT before performing the security operations, and updates the MQ descriptors as required. Therefore, besides the already existing components in modern processors, ProMT requires a small cache to cache the MQ descriptors and a persistent register to persist the hot MT root. On the other hand, Freij et al. [20] scheme, assumes a large pipeline and redundant hashing engines to update the MT root. Schemes like

Anubis [58] and Phoenix [11] assume a single persistent register. Thus, ProMT requires less hardware modifications when compared to schemes improving the MT root updates, but more hardware modifications when compared with recoverability schemes.

3.5.2 ProMT with large scale systems. In case of a large scale system with multiple memory controllers. We assume each memory controller to be responsible for the security operations of its associated memory region, where each memory region is protected with a separate integrity tree. Thus, each memory controller can implement ProMT locally to reduce the overheads of the MT root updates.

3.6 Security discussion

3.6.1 ProMT's impact on encryption. ProMT does not change the hot pages' encryption counters, but simply propagates the updates of these pages to the hot MT instead of the global MT. Thus, it has no effect on the encryption process. Note that using a different set of encryption counters would require enforcing a monotonically increasing encryption counter for the hot pages, which is required to prevent the reuse of encryption counters.

3.6.2 ProMT's impact on integrity. While ProMT maintains two separate integrity trees, ProMT does not change the integrity protection nor the verification process of the protected data. The integrity verification process can be split into two phases, the data integrity and the encryption counters integrity. The data integrity is verified using the HMAC, which is calculated over the data, the encryption counter, and the data cache line address. As ProMT does not affect any of the HMAC components, ProMT does not affect the data integrity verification. The encryption counter's integrity can be verified using the integrity tree, which hashes the encryption counter cache line along with other cache lines to generate the parent node. The integrity verification of both integrity trees is straightforward, but swapping pages and using different encryption counters in the hot MT might look as if it opens a room for known-plaintext attacks. Since the encryption counters protected by the hot MT are dynamically changing, an encryption counter of a smaller value might replace another encryption counter of a larger value. However, this replacement is still safe as the encryption counters in both cases belong to different data pages with different addresses, which will always generate different HMAC values as shown in Equation (1).

4 METHODOLOGY

In order to evaluate our scheme, we used the Gem5 [16] simulator, a cycle level simulator. To accurately measure the overheads of our scheme, we implemented the security metadata caches, the integrity trees, descriptors cache, modified the memory controller to handle the security metadata operations and ProMT operations, and added a latency of 24-cycles to simulate the overall AES encryption as in [55].

As shown in Table 1, we simulated a 4-core x86-64 processor with out-of-order execution. We used a 16GB PCM-based main memory with parameters as in [28]. We used three levels of caching where the LLC is shared. We used an 8-way, 256kB security metadata cache as in [11, 58].

Table 1: Configuration of the simulated system

Processor	
CPU	4 Cores, X86-64, Out-of-Order, 1.00GHz
L1 Cache	Private, 2 Cycles, 32KB,2-Way
L2 Cache	Private, 20 Cycles, 512KB, 8-Way
L3 Cache	Shared, 32 Cycles, 8MB, 64-Way
Cacheline Size	64Byte
DDR-based PCM Main Memory	
Capacity	16GB
PCM Latencies	Read 60ns, Write 150ns
Encryption Parameters	
Security Metadata Cache	256KB, 8-Way, 64B Block

We evaluated ProMT using 10 memory intensive workloads from SPEC2006 benchmark suite [23]. Additionally, we implemented and ran 7 multi-threaded persistent applications. We ran each application for 500M instructions after fast forwarding to a representative region and warming up the caches.

The in-house benchmarks were designed to stress the memory usage and were used in previous work [10, 35]. The description of these benchmarks is as below.
① ARSWP: Randomly swaps two keys in the database.
② RANDWR: Updates a random value in the database.
③ SEQWR: Updates the database in a sequential manner.
④ AVL: Maps the database into an AVL tree and a randomly generated key is searched in the mapped database. If the key is not found an insertion operation is triggered.
⑤ BTREE: Similar to AVL but uses a B-tree instead.
⑥ RBTREE: Similar to AVL but uses RBTREE instead. ⑦ QUEUE: Performs frequent enqueue and dequeue operations in a large queue.

Table 2 shows the used benchmarks, their MPKIs, and Writes Per Kilo Instruction (WPKI).

Table 2: Benchmarks Description

Benchmark	MPKI, WPKI	Benchmark	MPKI, WPKI
MCF	49.9, 8.64	ASTAR	0.001, 0.0005
Libquantum	27.1, 9.96	CACTUS	3.70, 0.93
LBM	23.9, 8.05	SJENG	0.48, 0.22
SOPLEX	2.13, 1.00	OMNETPP	0.49, 0.25
ZEUS	13.6, 4.41	BWAVES	25.5, 2.91
ARSWP	8.1, 3.1	QUEUE	69.2, 32.84
RBTREE	4.16, 1.60	BTREE	6.20, 2.38
RANDWR	19.30, 8.92	SEQWR	17.05, 7.8
AVL	0.04, 0.01		

The following schemes are used in our evaluation:
① **Lazy-Update:** Updates the encryption counters in the caches and does not propagate the update to the integrity tree until the dirty counter is evicted. This scheme does not ensure the system recoverability and is used as the **baseline**.
② **Strict-Update:** This scheme eagerly updates the integrity tree root and persists the updated nodes to the memory. Thus, it ensures an instant recovery of the system.
③ **ProMT:** Our proposed scheme that uses a Hot MT of 4 levels, and declares a hot page after 4 accesses to the page. This scheme eagerly updates the Hot MT and the global MT, and persists the updated nodes.
④ **ProMT-NP:** The optimization of our proposed scheme. This scheme works as the previous scheme, but does not persist the

updated Hot MT nodes, and relies on rebuilding the Hot MT during the recovery time.

5 EVALUATION

In this section, we evaluate ProMT against the schemes mentioned in 4. Then, we conduct a sensitivity analysis by varying the parameters of our scheme and show their impact on the system's performance.

5.1 The impact of ProMT on performance

Figure 8 shows the performance of the schemes normalized by the lazy update scheme. The Strict-Persist scheme incurs an average of 81.7% performance overhead, which spikes for the *LIBQUANTUM, LBM, RANDWR, RBTREE, ARSWP* and *Zeus* applications. While high overhead in the *SPEC* applications is caused by their high number of writes as shown in Table 2, the overhead of the parallel persistent applications is caused by there low security metadata cache hit rate.

Note that the Strict-Persist scheme overheads are directly related to the number of writes, as each write will result in a root update, which requires 11 MAC calculations. On the other hand, ProMT significantly improves the performance with an average overhead of 25.6%, which spikes for *LIBQUANTUM, RANDWR* and *LBM* applications. Note that ProMT reduced the performance overhead of these applications to 2.12x, 2.27x and 1.86x respectively.

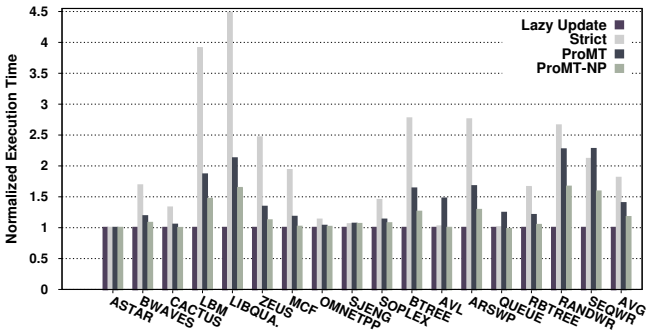

Figure 8: Normalized execution time of the evaluated schemes.

The performance improvement of ProMT can be attributed to its ability to detect the hot pages and mapping them into the hot MT, which leads to less number of MAC calculations to update the root. Figure 9 shows the normalized number of MAC calculations by the schemes. As shown in Figure 9, ProMT reduces the number of MAC calculations to an average of 2.4x, where the Strict-Persist scheme has an average of 29.13x MAC calculations.

As Figure 10 shows, *SJENG* and *ASTAR* have very low hit rate in the hot MT and the descriptors cache (35.5%, 48.7%, 35.9%, and 5.3%), which means these applications are detecting a few pages as hot but these pages are not getting reused, which can be inferred by comparing the number of MAC calculations for these applications in ProMT against the Strict-Persist scheme. In contrast, *LIBQUANTUM* and *LBM* applications are causing a high number of MAC calculations due to the high write intensity in these applications,

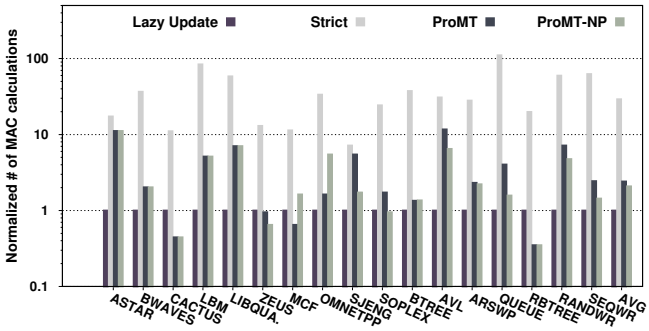

Figure 9: Normalized number of MAC calculations.

as shown in Table 2. Note the number of MAC calculations in these applications is slightly higher than the number of levels in hot MT, which is caused by misses and evictions from hot MT. Finally, we notice that *QUEUE* and *AVL* applications have better performance in the strict scheme than ProMT, despite having a high number of writes. Which is explained by the tremendous reduction in the number of memory reads due to the increased cache hit rate as shown later.

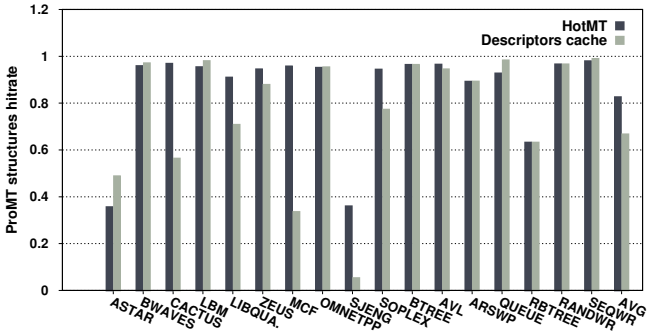

Figure 10: ProMT structures hit rates.

On the other hand, we note that *CACTUS*, *MCF*,*RBTREE*, and *ZEUS* applications are causing less MAC calculations than the baseline, which can be explained by the high number of hot pages in these applications (hot MT hit rate), and the high hit rates in the descriptors cache as shown in Figure 10. Additionally, these applications are showing the least hit rates in the security metadata cache hit rates (64.9%, 63.7%, 78.3% and 73.7%), which causes high number of evictions in the baseline and translates to additional number of MAC calculations.

After evaluating ProMT and analyzing the performance overheads of ProMT, we highlight that the overheads are mainly generated by the number of MAC calculations and holding the atomicity requirements, which requires holding all the data writes and the global/hot MT updates in the WPQ until the root is successfully updated. However, holding the writes in the WPQ can lead to stalling the processor due to an inability to evict dirty cached nodes as the WPQ is full. Thus, in ProMT-NP, we eagerly update the hot MT but without persisting the hot MT updates. While this can reduce the number of writes and alleviate some of the pressure of the WPQ, it

will require rebuilding the hot MT after crashes during the recovery phase. As shown in Figure 8, ProMT-NP reduces the performance overhead even further to reach 17.4%, which is dominated by the same applications as ProMT. However, ProMT-NP still drops the overhead of all the applications.

5.2 The impact of ProMT on the number of memory reads

ProMT improves the system's performance using a small hot MT. However, detecting the hot pages requires tracking the memory accesses to these pages, which is done using the MQ and cached in the descriptors cache. Thus, whenever a cache line is requested, its associated descriptor block needs to be updated. While updating the page's associated descriptor is typically done in the cache, a miss in the descriptors cache requires a memory read to fetch its associated descriptor. On the other hand, ProMT can reduce the number of the security metadata reads required to verify hot page's integrity.

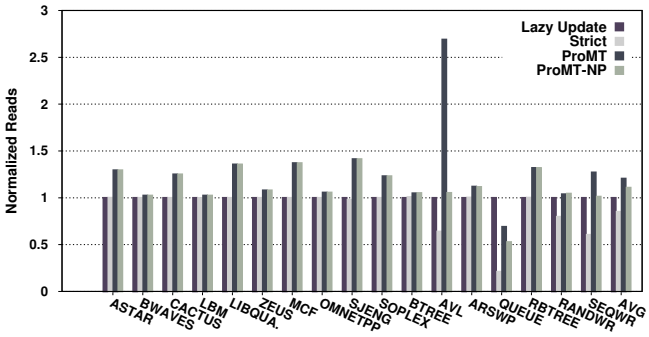

Figure 11: Normalized number of reads.

Figure 11 shows the impact of ProMT on the number of memory reads. As shown in Figure 11, ProMT causes an average of 20% memory reads, which is directly correlated with the descriptors cache hit rate shown in Figure 10. We observe the number of reads in *QUEUE* and *AVL* applications drops significantly in the strict persist scheme, which is caused by the increased cache hit rate. On the other hand, our scheme can affect the number of memory reads in two contradicting ways, the overheads of the descriptors cache misses dominates the effect of reads reduction for the integrity verification. Note that ProMT and ProMT-NP have the same impact on the number of reads, as the only difference is not pushing the hot MT updates to the WPQ. However, ProMT can adopt Synergy [43] and replace the ECC bits with the MAC value to reduce the number of reads (results are not shown for the lack of space), which drops the average number of reads from 120% to an average of 59.3%.

5.3 The impact of ProMT on the number of memory writes

Ensuring the memory integrity and the system's ability to recover from crashes requires maintaining a root value that reflects the memories most recent state. While having a fresh root enables the system recovery, rebuilding the integrity tree can take several hours as indicated by Anubis [58]. Thus, the integrity tree, or at least some of the tree levels, need to be persisted to ensure a fast recovery, as discussed in Triad-NVM [13]. However, strictly persisting the integrity tree nodes exacerbates the NVM's life endurance problem.

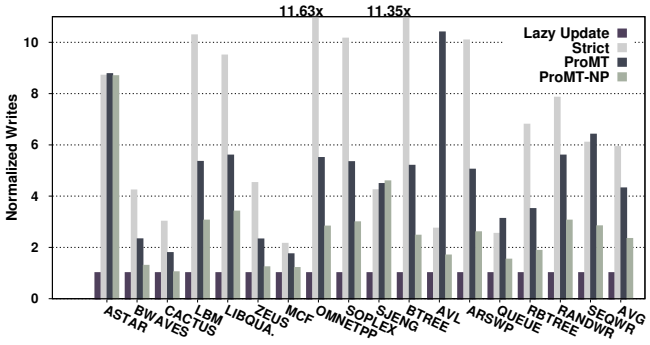

Figure 12: Normalized number of writes.

Figure 12 shows the impact of ProMT on the number of memory writes. As Figure 12 shows, our schemes incur an average of 3.8x and 2.33x writes for ProMT and ProMT-NP respectively. We observe that ProMT reduces the number of writes by 40-50% except for *ASTAR*, *SJENG*, *AVL*, and *MCF*. ProMT causes higher number of writes for *ASTAR* and *SJENG* applications, which are caused by detecting some pages as hot but barely using them, thus leading to extra number of writes when the pages are evicted from the hot MT, which can be explained by the low hit rates of these applications as shown in Figure 10. *MCF* shows an improvement of 20% only, which is due to the large working set that causes a large number of hot MT evictions. On the other hand, *AVL* shows a significant increase in the number of writes as it has a very small number of writes, thus the small increase caused by hot MT evictions are showing a large increase in the normalized figure.

Similarly, ProMT-NP reduces the number of writes by 65% to 75% except for *ASTAR, SJENG*, and *MCF*. *ASTAR* does not show any improvement nor extra overhead, as the writes to the hot MT are relaxed, which makes for the overheads of hot MT evictions, and *MCF* shows 45% reduction.

5.4 Sensitivity analysis

5.4.1 The impact of hot pages detection threshold on performance. The hot page detection threshold can affect the performance in two different ways. First, detecting the hot pages earlier by reducing the hot page detection threshold. Second, preventing the premature hot pages eviction by increasing the hot page detection threshold.

Figure 13 shows the average impact of varying the hot pages detection threshold on the performance. As shown in Figure 13, the performance overhead of ProMT remains at 40% when the threshold is below 4 accesses, but increases slightly to reach 43.1% and 44.8% when the number of accesses is increased to 8 and 16, respectively. However, we observe that increasing the threshold negatively impacts the performance of all the applications except for *MCF*, where the performance improves by 5.2%. This improvement is stemming from the prevention of premature hot pages eviction. Similarly, the performance overhead of ProMT-NP almost stays at

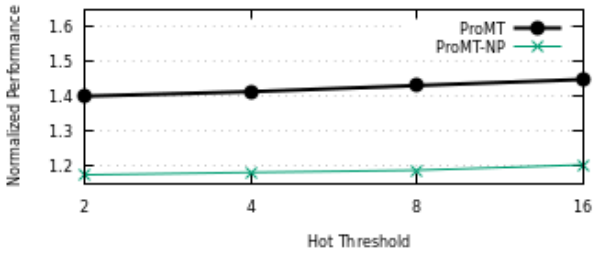

Figure 13: Performance sensitivity to the hot page detection threshold.

18% when the threshold is below 4, but increases to 18.7% and 20.2% when the threshold increases to 8 and 16, respectively.

5.4.2 The impact of hot MT levels on performance. The size of the hot region or the number of hot MT levels can affect the performance in two different ways. First, decreasing the number of hot MT levels/reducing the size of the hot region can lead to reducing the number of MAC calculations required to update the root. Second, increasing the number of hot MT levels/increasing the size of the hot region can lead to increasing the hit rate of hot pages.

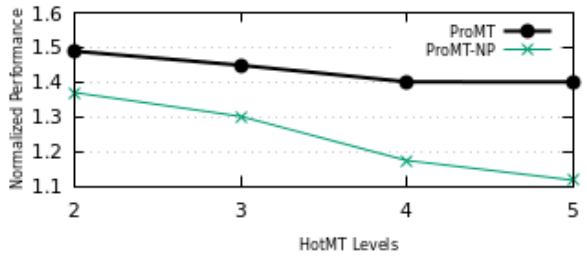

Figure 14: Performance sensitivity to the number of Hot MT levels.

Figure 14 shows the impact of varying the number of hot MT levels on the performance. As shown in Figure 14, the performance overhead of ProMT is reduced from 48.8%, 44.8%, and 40% when the number of levels is increased from 2, 3, to 4. Then, it starts to increase and reaches 40.02% when the number of levels reaches 5. However, we observe that increasing the number of hot MT levels negatively impacts the performance of all the applications except for *MCF*, where the performance improves from 3.01x, 2.32x, to 1.18x when the number of levels increases from 2, 3, to 4. However, the performance starts to decrease slightly when the number of levels is increased to 5. Additionally, the performance of *OMNETPP* improves by 4% when the number of levels increases from 2 to 3, but then increases again when the number of levels is increased to 4 and 5. Similar behavior is observed in *ZEUS, SOPLEX* and *SJENG*. This behavior can be explained by the applications having a working set that cannot be accommodated in the processor caches, but can be protected by the small hot MT, which the application is accessing frequently. The behavior of other applications indicates having a large working set that the hot MT cannot cover.

On the other hand, the performance of ProMT-NP improves as the number of levels increases as shown in Figure 14. The performance overhead of ProMT-NP is 36.9%, 30.1%, 17.4%, and 11.7% for hot MT levels of 2, 3, 4, and 5, respectively. We observe that all the applications are showing a similar behavior when the number of hot MT levels increases. This behavior can be explained by the large hot region of 128MB, which may accommodate a large portion of the memory footprint of these applications. However, increasing the number of hot MT levels increases the recovery time significantly as discussed in [58].

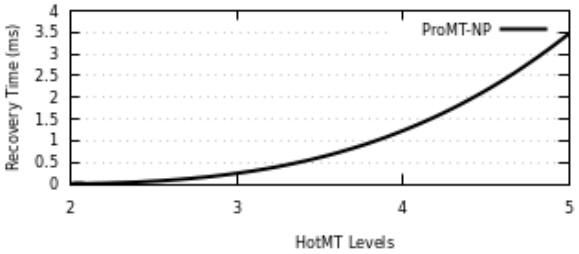

Figure 15: ProMT-NP recovery time.

5.4.3 The impact of hot MT levels on ProMT-NP recovery time. ProMT does not require any recovery time as the hot MT as eagerly updated and persisted. On the other hand, ProMT-NP improves the performance by updating the hot MT nodes in the cache but not the memory.

Figure 15 shows the impact of the number of hot MT levels on the recovery time of ProMT-NP. Increasing the number of the hot MT levels can lead to an exponential increase in the recovery time, as it would take 0.5 µs to recover a small hot MT of 3 levels; this recovery time grows to reach 3.5 ms when the number of hot MT levels is increased to 5.

6 RELATED WORK

In this section, we review the related studies in secure memory implementation, NVM crash consistency, and improving the performance of NVMs.

Secure memory implementation has been attracting the attention of researchers in the past few years. Synergy [43] proposed using the Error Correction Code (ECC) chip to store the data MAC values, and store the ECC bits in a different location in the memory. Synergy reduces the number of memory reads by fetching the MAC alongside the data in a single read, and only reads the ECC bits when an error is detected, which can be detected using the MAC. VAULT [47] proposed a variable arity integrity tree, which aims to reduce the integrity tree levels and improve its cacheability. Similarly, Morphable [42] packs more encryption counters in a single cache line, which results in a smaller integrity tree and increases its cacheability. Taassori et al. [46] proposed using separate integrity trees and separate security metadata caches for each application to prevent side-channel attacks in the security metadata cache. Additionally, the proposed scheme combines the parity bits inside the integrity tree structure to reduce the memory accesses in case of errors.

In terms of crash consistency, Osiris [55] proposed a scheme that allows the encryption counters recovery with a minimal performance overhead, but assumes its feasible to rebuild the integrity tree after crashes. Anubis [58] argues that rebuilding the integrity tree is not always possible, and it can take several hours for practical size NVMs. Anubis proposed a scheme that enables recovering the system in a fraction of a second by tracking the security metadata cache updates. Phoenix [11] argues that recovering the security metadata cache does not require tracking all the ToC updates, and proposed a scheme to recover the security metadata cache of ToC integrity protected system with minimal overheads. SuperMem [60] proposed using a write-through security metadata cache that utilizes a locality-aware counter write coalescing scheme. Several other studies addressed the crash consistency problem [13, 14, 34, 53, 54, 56, 59]. Additionally, different aspects of NVM performance, security, and integration are discussed in various studies [9, 10, 17, 18, 21, 24, 45, 57].

7 CONCLUSION

In this work, we present a novel mechanism that improves the performance of integrity protected systems by reducing the number of MAC calculations required to update the integrity tree root. Maintaining an up-to-date root is essential for the system's recoverability. We note that the memory pages are updated at different rates, and the application's memory footprint does not consume the whole memory. Thus, we proposed ProMT, which reduces the number of MAC calculations by using a small integrity tree that protects the application's hot pages. Then, we optimized ProMT even further and proposed ProMT-NP, which relaxes persisting the hot MT nodes to achieve better performance at the cost of a fraction of a second in the recovery time.

8 ACKNOWLEDGMENTS

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Approved for public release. Distribution is unlimited.

REFERENCES

[1] [n.d.]. AMD Memory Encryption. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf. Accessed: 2020-02-15.

[2] [n.d.]. deprecating PCOMTI. https://software.intel.com/content/www/us/en/develop/blogs/deprecate-pcommit-instruction.html. Accessed: 2020-03-03.

[3] [n.d.]. Enhancing High-Performance Computing with Persistent Memory Technology. https://software.intel.com/content/www/us/en/develop/articles/enhancing-high-performance-computing-with-persistent-memory-technology.html. Accessed: 2020-02-15.

[4] [n.d.]. Intel Architecture, Memory Encryption Technologies Specification. https://software.intel.com/sites/default/files/managed/a5/16/Multi-Key-Total-Memory-Encryption-Spec.pdf. Accessed: 2020-02-15.

[5] [n.d.]. Intel Optane DC Persistent Memory. https://builders.intel.com/docs/networkbuilders/intel-optane-dc-persistent-memory-telecom-use-case-workloads.pdf. Accessed: 2020-24-07.

[6] [n.d.]. NVDIMM. https://www.jedec.org/sites/default/files/Bill_Gervasi.pdf. Accessed: 2020-02-02.

[7] [n.d.]. NVDIMM - Changes are Here So What's Next. https://www.snia.org/sites/default/files/SSSI/NVDIMM%20-%20Changes%20are%20Here%20So%20What's%20Next%20-%20final.pdf. Accessed: 2020-03-29.

[8] [n.d.]. NVDIMM-P. https://software.intel.com/content/www/us/en/develop/articles/enabling-persistent-memory-in-the-storage-performance-development-kit-spdk.html. Accessed: 2020-02-02.

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

[10] Mazen Alwadi, Vamsee Reddy Kommareddy, Clayton Hughes, Saimon Hammond, and Amro Awad. 2021. Stealth-Persist: Architectural Support for Persistent Applications in Hybrid Memory Systems. In *2021 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. ACM.

[11] Mazen Alwadi, Aziz Mohaisen, and Amro Awad. 2019. Phoenix: Towards Persistently Secure, Recoverable, and NVM Friendly Tree of Counters. arXiv:1911.01922 [cs.CR]

[12] Amro Awad, Pratyusa Manadhata, Stuart Haber, Yan Solihin, and William Horne. 2016. Silent shredder: Zero-cost shredding for secure non-volatile main memory controllers. *ACM SIGOPS Operating Systems Review* 50, 2 (2016), 263–276.

[13] Amro Awad, Yan Solihin, Laurent Njilla, Mao Ye, and Kazi Zubair. 2019. Triad-NVM: Persistency for Integrity-Protected and Encrypted Non-Volatile Memories. In *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 169–180.

[14] Amro Awad, Suboh Suboh, Mao Ye, Kazi Abu Zubair, and Mazen Al-Wadi. 2019. Persistently-Secure Processors: Challenges and Opportunities for Securing Non-Volatile Memories. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 610–614.

[15] Amro Awad, Yipeng Wang, Deborah Shands, and Yan Solihin. 2017. Obfusmem: A low-overhead access obfuscation for trusted memories. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 107–119.

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

[17] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. 2012. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM Sigplan Notices* 47, 4 (2012), 105–118.

[18] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*. 271–282.

[19] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016, 086 (2016), 1–118.

[20] Alexander Freij, Shougang Yuan, Huiyang Zhou, and Yan Solihin. [n.d.]. Persist Level Parallelism: Streamlining Integrity Tree Updates for Secure Persistent Memory. ([n. d.]).

[21] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Pisces: a scalable and efficient persistent transactional memory. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 913–928.

[22] Shay Gueron. 2016. A Memory Encryption Engine Suitable for General Purpose Processors. Cryptology ePrint Archive, Report 2016/204. https://eprint.iacr.org/2016/204.

[23] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (sep 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[24] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M Chen, Satish Narayanasamy, and Thomas F Wenisch. 2017. Language-level persistency. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 481–493.

[25] Vamsee Reddy Kommareddy, Amro Awad, Clayton Hughes, and Simon David Hammond. 2019. *Enforcing Fairness in Disaggregated Non-Volatile Memory Systems*. Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

[26] Vamsee Reddy Kommareddy, Clayton Hughes, Simon David Hammond, and Amro Awad. 2020. DeACT: Architecture-Aware Virtual Memory Support for Fabric Attached Memory Systems. *arXiv preprint arXiv:2008.00171* (2020).

[27] Vamsee Reddy Kommareddy, Jagadish Kotra, Clayton Hughes, Simon David Hammond, and Amro Awad. [n.d.]. PreFAM: Understanding the Impact of Prefetching in Fabric-Attached Memory Architectures. ([n. d.]).

[28] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. *ACM SIGARCH Computer Architecture News* 37, 3 (2009), 2–13.

[29] Benjamin C Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. 2010. Phase-change technology and the future of main memory. *IEEE micro* 1 (2010), 143–143.

[30] Benjamin C Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. 2010. Phase-change technology and the future of main memory. *IEEE micro* 30, 1 (2010), 143–143.

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

[32] Zhongqi Li, Ruijin Zhou, and Tao Li. 2013. Exploring high-performance and energy proportional interface for phase change memory systems. *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)* (2013), 210–221.

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

[34] Sihang Liu, Aasheesh Kolli, Jinglei Ren, and Samira Khan. 2018. Crash consistency in encrypted non-volatile main memory systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 310–323.

[35] Sihang Liu, Korakit Seemakhupt, Gennady Pekhimenko, Aasheesh Kolli, and Samira Khan. 2019. Janus: Optimizing memory and storage support for non-volatile memory systems. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 143–156.

[36] Thomas S Messerges. 2000. Securing the AES finalists against power analysis attacks. In *International Workshop on Fast Software Encryption*. Springer, 150–164.

[37] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual international symposium on Computer architecture*. 24–33.

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

[39] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. 2007. Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 183–196.

[40] Brian Rogers, Chenyu Yan, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. 2008. Single-level integrity and confidentiality protection for distributed shared memory multiprocessors. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. IEEE, 161–172.

[41] Andy Rudoff. 2017. Persistent memory programming. *Login: The Usenix Magazine* 42, 2 (2017), 34–40.

[42] Gururaj Saileshwar, Prashant Nair, Prakash Ramrakhyani, Wendy Elsasser, Jose Joao, and Moinuddin Qureshi. 2018. Morphable Counters: Enabling Compact Integrity Trees For Low-Overhead Secure Memories. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 416–427.

[43] Gururaj Saileshwar, Prashant J Nair, Prakash Ramrakhyani, Wendy Elsasser, and Moinuddin K Qureshi. 2018. Synergy: Rethinking secure-memory design for error-correcting memories. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 454–465.

[44] G Edward Suh, Dwaine Clarke, Blaise Gasend, Marten Van Dijk, and Srinivas Devadas. 2003. Efficient memory integrity verification and encryption for secure processors. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* IEEE, 339–350.

[45] Shivam Swami and Kartik Mohanram. 2018. ARSENAL: Architecture for secure non-volatile memories. *IEEE Computer Architecture Letters* 17, 2 (2018), 192–196.

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

[47] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. 2018. 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*. ACM, 665–678.

[48] Rujia Wang, Youtao Zhang, and Jun Yang. 2017. Cooperative Path-ORAM for Effective Memory Bandwidth Sharing in Server Settings. In *High Performance Computer Architecture (HPCA)*.

[49] Yao Wang, Andrew Ferraiuolo, and G Edward Suh. 2014. Timing channel protection for a shared memory controller. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 225–236.

[50] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. Hikv: A hybrid index key-value store for dram-nvm memory systems. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 349–362.

[51] Jian Xu and Steven Swanson. 2016. {NOVA}: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*. 323–338.

[52] Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. 2006. Improving cost, performance, and security of memory encryption and authentication. In *ACM SIGARCH Computer Architecture News*, Vol. 34. IEEE Computer Society, 179–190.

[53] Fan Yang, Youmin Chen, Haiyu Mao, Youyou Lu, and Jiwu Shu. 2020. ShieldNVM: An Efficient and Fast Recoverable System for Secure Non-Volatile Memory. *ACM Transactions on Storage (TOS)* 16, 2 (2020), 1–31.

[54] Fan Yang, Youyou Lu, Youmin Chen, Haiyu Mao, and Jiwu Shu. 2019. No compromises: Secure NVM with crash consistency, write-efficiency and high-performance. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[55] Mao Ye, Clayton Hughes, and Amro Awad. 2018. Osiris: A Low-Cost Mechanism to Enable Restoration of Secure Non-Volatile Memories. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 403–415.

[56] Mao Ye, Kazi Zubair, Aziz Mohaisen, and Amro Awad. 2019. Towards Low-Cost Mechanisms to Enable Restoration of Encrypted Non-Volatile Memories. *IEEE Transactions on Dependable and Secure Computing* (2019).

[57] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P Jouppi. 2013. Kiln: Closing the performance gap between systems with and without persistence support. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 421–432.

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

[59] Pengfei Zuo and Yu Hua. 2018. SecPM: a secure and persistent memory system for non-volatile memory. In *10th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*.

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