ShieldNVM: An Efficient and Fast Recoverable System for Secure Non-Volatile Memory

FAN YANG, YOUMIN CHEN, HAIYU MAO, YOUYOU LU, and JIWU SHU,
Tsinghua University

Data encryption and authentication are essential for secure non-volatile memory (NVM). However, the introduced security metadata needs to be atomically written back to NVM along with data, so as to provide crash consistency, which unfortunately incurs high overhead. To support fine-grained data protection and fast recovery for a secure NVM system without compromising the performance, we propose ShieldNVM. It first proposes an epoch-based mechanism to aggressively cache the security metadata in the metadata cache while retaining the consistency of them in NVM. Deferred spreading is also introduced to reduce the calculating overhead for data authentication. Leveraging the ability of data hash message authentication codes, we can always recover the consistent but old security metadata to its newest version. By recording a limited number of dirty addresses of the security metadata, ShieldNVM achieves fast recovering the secure NVM system after crashes. Compared to Osiris, a state-of-the-art secure NVM, ShieldNVM reduces system runtime by 39.1% and hash message authentication code computation overhead by 80.5% on average over NVM workloads. When system crashes happen, ShieldNVM's recovery time is orders of magnitude faster than Osiris. In addition, ShieldNVM also recovers faster than AGIT, which is the Osiris-based state-of-the-art mechanism addressing the recovery time of the secure NVM system. Once the recovery process fails, instead of dropping all data due to malicious attacks, ShieldNVM is able to detect and locate the area of the tampered data with the help of the tracked addresses.

CCS Concepts: • **Security and privacy** → **Security in hardware**; **Systems security**; *Database and storage security*; • **Computer systems organization** → **Architectures**; • **Hardware** → *Emerging technologies*;

Additional Key Words and Phrases: Non-volatile memory, crash consistency, memory security

ACM Reference format:
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 Trans. Storage* 16, 2, Article 12 (May 2020), 31 pages.
https://doi.org/10.1145/3381835

1 INTRODUCTION

Emerging byte-addressable non-volatile memory (NVM) technologies, such as PCM, STT-RAM, ReRAM, and Intel 3D XPoint, bring massive amounts (up to 6 terabytes (TB) per two-socket system)

This work was supported by National Key Research and Development Program of China (grant no. 2018YFB1003301), the National Natural Science Foundation of China (grant no. 61832011), and the Research and Development Plan in Key Field of Guangdong Province (grant no. 2018B010109002).

Authors' addresses: F. Yang, Y. Chen, H. Mao, Y. Lu (corresponding author), and J. Shu (corresponding author), Tsinghua University; emails: {yangf17, chenym16, mhy15}@mails.tsinghua.edu.cn, {luyouyou, shujw}@tsinghua.edu.cn.

1553-3077/2020/05-ART12 $15.00
https://doi.org/10.1145/3381835

of byte-addressable NVM at speeds close to those of dynamic random access memory (DRAM) for a fraction of DRAM's cost, making them a promising alternative to DRAM as the main memory [1, 29, 30]. Further, these new memory technologies bring data durability to the in-memory system, which blurs the difference between storage and memory, making it possible to store and manipulate persistent data directly in memory rather than relying on software intermediaries (e.g., the file system). Thus, the application start-up time is reduced because there is no need to recreate memory data structures.

Since NVM is directly attached to the memory bus (a.k.a. persistent memory), it is vulnerable to malicious attacks similar to that from which the DRAM system may suffer. There are mainly two types of attack models: ① data confidentiality attack (i.e., stealing the privacy data) and ② data integrity attack (e.g., data spoofing, splicing, and replaying). In response to the data confidentiality attack, state-of-the-art secure memory systems employ counter-mode encryption (CME) [34, 57, 59, 65, 72]. In CME, each memory block (cache line) is associated with a unique counter that is used along with the key to encrypt the data block once the data block is evicted out of the CPU cache, then the specified counter increases by one. In response to integrity attack, the Bonsai Merkle Tree (BMT) is a widely used Merkle Tree (MT) authentication in current secure architecture [6, 46, 49, 58, 60, 66]. It is based on CME and builds up the hash tree over counters instead of the whole memory data. To prevent spoofing and splicing attacks, the BMT builds a single layer of hash message authentication codes (data HMACs) for the memory data. It also builds a hierarchical tree structure of counter HMACs over counters to detect replay attacks.

In the legacy DRAM main memory, data in DRAM are quickly lost when crashes/reboots happen due to the volatility. Thus, memory data remanence and recovery are not considered. However, since NVMs retain data even after power loss, memory data remanence is a real security vulnerability. NVMs also enables persistent applications that can recover after a crash or reboot by keeping data persisted in memory. Therefore, the system should be able to recover securely and guarantee its data integrity and confidentiality across crashes/reboots, which means that necessary measures for security and persistence of security metadata (e.g., counters and HMACs) need to be taken throughout the lifetime of the system and across system reboots/crashes [6, 66, 67, 70].

To improve performance, traditional DRAM-based secure memory systems store these frequently accessed security metadata in dedicated cache space [31, 34, 51] or directly store them in the last-level cache [15, 49]. Further, by leveraging the secure property of on-chip registers and caches in CPUs, the verification of a memory block only needs to proceed up the MT until the first cached node is found. Thus, it is not necessary to fetch and verify all MT nodes up to the tree root on each memory access, significantly reducing memory bandwidth consumption and improving verification performance. However, metadata caching is non-trivial for secure NVM. Although placing the frequently accessed security metadata into the on-chip cache improves performance, it causes consistency problems for NVM. The cached data in the CPU cache may be lost after a system/power failure, thus leading to the corrupted state when the data reaches NVM while the relevant metadata (e.g., encryption counter and updated MT nodes) is not reflected in NVM. Such inconsistency between data and metadata can prevent us from correctly decrypting and authenticating the NVM data. For example, if the system fails after the encrypted data is persisted into NVM, but before its counter has been persisted, the persisted data cannot be decrypted correctly without the correct counter. Consequently, after recovery, the system will have an inconsistency between the versions of data and its corresponding security metadata, leading to the failure of data decryption and authentication.

As a result, how to achieve crash consistency becomes the building block for secure NVM systems. Specifically, it is essential to guarantee ❶ consistency between data and security metadata and ❷ the internal consistency of the MT simultaneously. However, implementing such a crash consistency mechanism is challenging. First, data write-backs result in updating the associated counter and all levels up to the tree root. To ensure correct recovery after crashes/reboots, such updates need to occur atomically. Second, updating all levels up to the tree root prolongs the write latency, and incurs extra HMAC computation overhead and NVM write traffic. Considering the limited write-endurance and runtime performance, it is impractical to update all levels of the MT on each data write-back. Third, to continue security support for the NVM data after crashes/reboots, security metadata needs to be recovered. Hence, the design of secure NVM systems should consider the recoverability of the encryption and authentication. However, NVM capacities are expected to reach up to 6 TB per two-socket system [7], and such a recovery process can take hours, which cannot meet the availability requirement that needs fast recovery.

We have implemented a naive approach called *strict persistence*, which ensures atomicity by aggressively writing back all related cached security metadata. Strict persistence would eliminate the consistency issues of the secure metadata in secure NVM. However, it deteriorates the system performance by 62.8%, increases memory write traffic by 5.5×, and increases HMAC computation overhead by 8.3× in comparison to the NVM-based secure system without consistency guarantee. The root cause of such inefficiency lies in two aspects. The first aspect is *write amplification*. Each time a data block is evicted, the related encryption counters and all tree nodes in the MT (can be tens of levels) need to be flushed to NVM as well. The second aspect is *cascading calculating on the MT*. Once a counter is incremented, all of its ancestors up to the root node in the MT need be recalculated, increasing HMAC computation overhead.

In addition, the calculation of each HMAC in the BMT must be executed one after another (instead of in parallel), as the parent node stores several counter HMACs, and each of them is generated with one of its child nodes. Thus, the write-back should wait until the root is updated, prolonging the NVM write latency. Further, crash-consistent software for NVM usually uses cache flush instructions (e.g., clwb, sfence) to ensure the ordering of writes and to guarantee the durability of data [12, 13, 20, 45, 62, 64], making data recoverable on NVMs. This kind of instruction places writes to the NVM (or more specifically, the persistent domain) on the critical path of the program execution [35]. The cascading calculating on the MT further degrades the applications' performance by increasing the latency of the write operations. Although the strict persistence achieves low recovery time (because it strictly ensures the consistency between data and associated security metadata), it incurs significant runtime overhead, such as reducing the system performance, increasing the number of writes to NVM, and generating more HMAC computation overhead.

SCA [34] and SuperMem [72] address the crash consistency in encrypted NVM systems. However, they did not consider the crash consistency issue in authenticated NVM. Osiris, Arsenal, Stash, and Anubis [58, 60, 67, 70] propose that the MT can be successfully reconstructed as long as the data block, the corresponding counter in the NVM (i.e., leaf node), and the root node in the trusted computing base (TCB) are updated consistently. Therefore, the inner nodes of the MT do not need to be flushed out of the cache for each evicted data block. However, since the data block and the root node need to be updated atomically, the evicted data will be blocked until the root node is updated. Although this method reduces lots of writes to NVM, the aforementioned performance and HMAC computation overhead still exists. In addition, this approach ensures that the replay attacks can always be detected if the reconstructed root node mismatches with the root node stored in the TCB (a non-volatile register). However, it is unable to pinpoint the exact data block that has been corrupted and thus causes a single point of failure: if a replay attack happens, it is possible that all of the data HMAC match with their corresponding data block and counter, but the reconstructed root node mismatches with the TCB one [6, 66]. As a result, all of the data in NVM should be dropped once an integrity attack occurs because none of the memory is integrity verifiable. Triad-NVM [6] proposes persisting the first N levels of the MT to enable high resolution of identifying unverifiable locations. However, it cannot totally avoid this problem. As for the recovery, Osiris [67] introduces extra ECC bits to correctly recover the counters even when they are lost. However, before completing the first memory access, counter recovery schemes need to fix the counters and then build up the whole MT upon all of the counters, hence iterating over a huge number of memory blocks. Therefore, Osiris would take hours to recover the systems after a crash [70] for TB-scale NVM. Anubis [70] addresses the recovery time in secure NVM systems by tracking the addresses of metadata caches blocks in NVM. And it improves Osiris's recovery time significantly. However, it needs to reserve spaces in NVM for tracking and protect the integrity of these spaces. In addition, the overhead of updating up to the root and single point of failure still exists. In conclusion, some existing approaches are vulnerable to integrity attacks [34, 72]. For those that implement integrity verification, they cannot eliminate the aforementioned performance overhead caused by additional HMACs computing and pick out the tampered data after system failures [6, 58, 60, 67, 70]. The state-of-the-art solution, cc-NVM [66], did not consider and discuss the recovery time with practical NVM capacities.

To solve the preceding issues in secure NVM systems, and make secure protection efficient in NVM systems, we propose ShieldNVM. ShieldNVM is capable of fast recovery for encrypted and authenticated NVM systems while introducing minimal runtime crash consistency overhead. In addition, it is able to detect and locate the attacks in NVM both at runtime and after system crashes, minimizing the unverifiable locations. ShieldNVM is based on three key observations. First, data HMACs can detect and recover the stale counters. Due to the ability of data HMACs, we are enabled to lazily flush encryption counters to NVMs. In the absence of spoofing/splicing attacks, any stale counters after a system failure can be safely recovered. Second, adjacent data blocks evicted by CPU cache share the same ancestors of tree nodes in the MT, and tree nodes of higher tree level cover more data blocks. Due to this observation, we then propose an epoch-based consistent MT. It aggressively caches the tree nodes in metadata cache and atomically synchronizes the updates to NVM. As a result, the MT in NVM is consistently transitioned from an old state to a new state. Leveraging the consistency property of the MT in NVM, the ability to thwart replay attacks still holds. To reduce the HMAC calculating overhead, similar to the existing DRAM-based secure system, ShieldNVM calculates the new tree nodes from the bottom up and stops if it has already been cached. Instead, the root node is lazily updated only at synchronization (end of the epoch). Third, only the dirty security metadata needs to be recovered. Although the clean data in caches are consistent with the data in memory, the dirty data in cache is newer than that in memory. In ShieldNVM, the number of dirty metadata cache lines is limited to the number of the entries in the write pending queue (WPQ) (which is usually 64) in the memory controller. Through recording the addresses of these dirty security metadata, ShieldNVM can achieve fast recovery and locating the attacked data after crashes/reboots.

To evaluate the performance overhead of our proposed solution, we use Gem5 [8], a cycle-level simulator to run NVM-optimized workloads [34]. Our evaluations show that, on average, ShieldNVM increases the performance by 73.49%, reduces write traffic by 73.51%, and reduces HMAC computation overhead by 80.51% compared to strict persistence. Most importantly, ShieldNVM achieves a recovery time of about 0.0022 seconds, whereas Osiris requires an average of about 27 seconds for 16 GB to recover both encryption counters and the MT. Notably, the recovery time of ShieldNVM is only a function of the number of entries in the WPQ and does not increase linearly with memory size as in other schemes.

To summarize, our contributions are as follows:

- We propose a novel solution to eliminate the crash consistency overhead in secure NVM systems by deferring the HMAC computation to the end of the epoch. First, we present an epoch-based consistent BMT. It aggressively caches the frequently accessed security metadata in the metadata cache and atomically commits these updates to the persistent

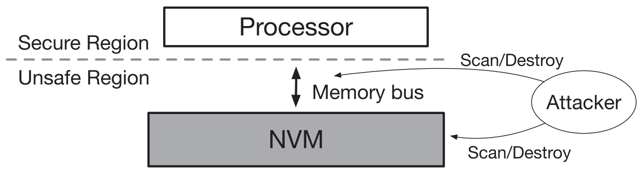

Fig. 1. Threat model.

domain. Second, we show that the HMAC computation process can stop up to the first cached MT node, which reduces a large amount of latency on the critical path of the writes caused by cascading HMAC computation. With such an epoch mechanism, ShieldNVM fully exploits the benefits of metadata caching with minimal crash consistency overhead.

- We use a small component called *Drainer* in the memory controller to record the addresses of the dirty metadata for both MT state transformation and fast recovery. ShieldNVM achieves fast recovery, which is beneficial for secure NVM systems that require high availability.
- We propose ShieldNVM, a novel hardware-only solution that enables low performance overhead, low write traffic, and low HMAC computation overhead for encrypted and authenticated NVM systems. ShieldNVM achieves fast recovery and not only detects attacks but also locates the tampered area even after a system failure.

The rest of the article is organized as follows. First, in Section 2, we provide a brief background on memory encryption, authentication, crash consistency problems, and new issues on each secure method. Section 3 presents quantitative results that demonstrate the problem. In Section 4, we depict the epoch-based consistent BMT and its optimization. Section 5 presents the evaluation methodology, and Section 6 presents our evaluation for ShieldNVM. Section 7 discusses related work. We conclude in Section 8.

2 BACKGROUND

2.1 Threat Model

Our threat model identifies two regions of a system just as in prior studies on hardware-based memory encryption and authentication. The secure region, the TCB, consists of the processor chip and core parts of the operating system (e.g., security kernels) shown in Figure 1. Any on-chip code or data (i.e., in registers or caches) is considered safe and cannot be observed or manipulated by attackers. The non-secure region includes all off-chip resources, primarily including the off-chip memory and the processor-memory bus [5, 32, 49, 55, 65]. The attacker can obtain secret values stored in memory or transferred through off-chip interconnects when the program is running, or directly steals the NVM DIMM (i.e., *data confidentiality attacks*). The adversary can also act as a man-in-the-middle to modify values in the physical memory and on all off-chip interconnects. This is called *data integrity attacks*, which include tampering the value directly (spoofing), exchanging the contents of two different memory addresses (splicing), and replaying the data to their old version (replay).

2.2 Memory Encryption

Memory encryption is done to ensure that the adversary cannot obtain any meaningful data stored outside the TCB. Therefore, any data evicted out of the CPU cache should be encrypted before being stored in memory. While the memory reads are in the critical path of program execution, memory encryption causes the high decryption latency after each memory read due to serial execution, as shown in Figure 2(a), thus significantly degrading system performance. CME is

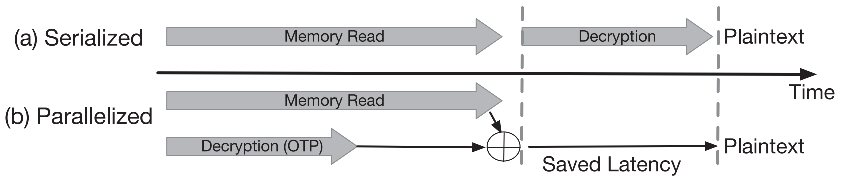

Fig. 2. Reduced latency during a read operation in CME.

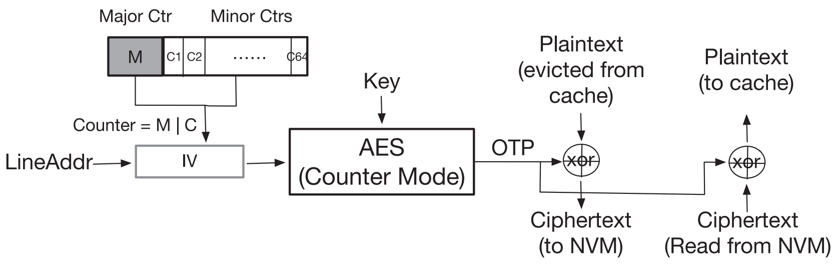

Fig. 3. State-of-the-art CME.

proposed to move the decryption latency out of the critical path of memory reads and hence has been widely used in encrypted memory systems [5, 11, 49, 55, 59, 68]. CME encrypts a data block by XORing it with a one-time pad (OTP). Specifically, the OTP is generated with a secret key and an initialization vector (IV) as inputs (the IV consists of the address of the data block and an associated counter). Conversely, when a data block is fetched from memory, the same IV is encrypted to generate the same OTP, and then we use it to decrypt the data block. Figure 3 shows the encrypt/decrypt process of CME. The main idea is to compute the OTP in parallel with the memory read, and then XOR the OTP with the ciphertext data to generate the plaintext, thus hiding the decryption latency, as shown in Figure 2(b).

The read/write granularity between the memory and cache is the cache-line block. The size of one cache-line block is usually 64 bytes. It is common to associate each cache block with a counter. The state-of-the-art schemes organize counters into major and minor counters (called a split-counter scheme) [65], where a small minor counter, typically 7 bits, is associated with each data cache block (64 B), and a larger major counter, typically 64 bits, is shared across all cache blocks of the same page (4 KB). Since the minor counter has only 7 bits, it overflows after 128 updates. When a minor counter overflows, the major counter is incremented by one, and all minor counters are reset. This requires additional memory reads and writes to re-encrypt all data cache blocks associated with the major counter.

The security of CME is based on the premise that each OTP is never reused for data encryption as it enables known plaintext attacks. This scheme is ensured to be secure by guaranteeing the uniqueness of each IV: (1) different data blocks are mapped to different counters, ensuring the spatial unique, and (2) the counter is increased by one for each data write-back (encryption), ensuring the temporal unique. Figure 3 depicts how IVs are established from counter blocks. Note that counters are packed in 64-B counter blocks. Each counter block contains 64 minor counters and a major counter. For the rest of the article, we use the split-counter scheme for the general memory encryption scheme.

2.3 Memory Authentication

Memory authentication is devoted to guaranteeing that a value loaded out of the TCB by a processor is equal to the most recent value that the processor writes. Spoofing and splicing attacks can be

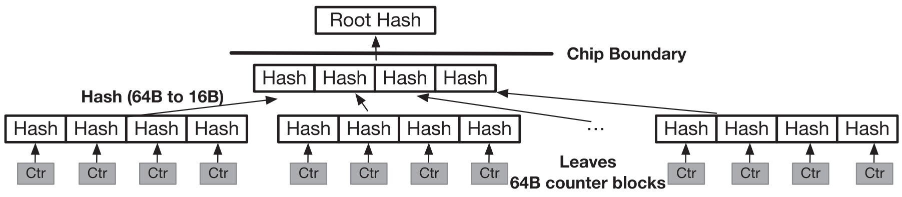

Fig. 4. Organization of the MT.

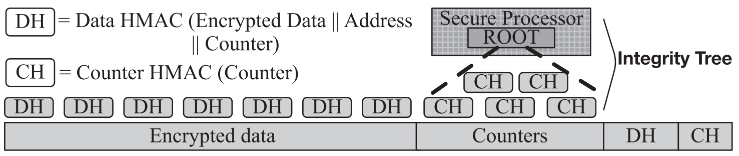
DH = Data HMAC (Encrypted Data || Address || Counter)
CH = Counter HMAC (Counter)
Secure Processor
ROOT
Integrity Tree
CH
DH
Encrypted data
Counters
DH
CH

Fig. 5. Data layout of the BMT.

detected by computing and writing back the HMACs for every evicted data, and the data is verified when it is read from memory. However, single-level HMAC over the whole memory is not robust enough because replay attacks can be applied by rolling back both the data and its HMAC to their old values. MT verification is a widely known way to detect replay attacks [15, 55]. An MT maintains hierarchical HMACs organized as a tree, with the memory data as its leaf nodes. A parent HMAC protects multiple child HMACs, and the fetched memory blocks are verified by checking its chain of HMAC values up to the root HMAC. The corresponding HMAC values of the tree are updated when a leaf node is evicted to memory. The root of the tree containing the whole memory blocks' information is stored on-chip and cannot be damaged or replayed by adversaries, which guarantees the integrity of any value in the memory.

Figure 4 depicts a general MT scheme that builds on the hashes over hashes. In this scheme, a number of cache blocks are hashed together using a cryptographic hash function stored in the processor. These hashes are hashed over and over again, and a tree structure is formed with the 64-B blocks being the leaves and one 64-B hash at the top of the tree as the root. Each level of the tree consists of hash values calculated over a number of its direct children nodes (those on the lower level). For instance, in each level, each node is 64 B, which consists of four 16-B hash values. Each 16-B hash value has been calculated over a 64-B node of the lower level. Thus, the tree is called a *4-ary tree* (each of the four nodes will have one parent node in the upper level). MTs have been considered thus far the most secure scheme with low on-chip storage overhead, because only the root needs to be kept inside the processor chip. However, since it builds up over the whole memory data, tree nodes can cause significant memory storage overhead.

The state-of-the-art memory authentication architecture is the BMT [6, 46, 49, 67] (Figure 5), which is deployed on the CME scheme. It first uses a single layer of message authentication codes (a.k.a. data HMACs) to detect spoofing and splicing attacks, where each HMAC is a hashed value generated by taking the encrypted data block, the counter, and its address as inputs. It then builds the MT, a hierarchical tree structure, over the counters to prevent replay attacks. Each leaf node of the MT is a memory line of counters, and the parent nodes (a.k.a. counter HMACs) store the HMACs of its children computed using a keyed hashing function (e.g., HMAC based on SHA-1). The secret HMAC key and the root of the MT are stored in secure on-chip registers to prevent the MT from being replayed from the bottom up. The BMT is different from the traditional approaches that build the MT over both counters and data [15]. In the BMT, the encrypted data blocks are not directly protected. However, it is immune to replay attacks since the data HMACs include the

MT-protected counters as input. Compared to the MT built over the whole memory, the BMT has lower metadata storage overhead and shorter tree depth.

2.4 Caching Security Metadata

Caching security metadata can boost performance for both CME and the BMT [15, 49, 54]. For example, take a memory read and decryption. If the corresponding counter (in CME) has already been cached, the OTP generation and the read access can be executed in parallel, hiding the OTP generation latency. Typically, those counters of different data blocks in the same data page (i.e., 4 KB) are organized into the same cache line [34, 49, 65]. The accesses to these continuous data blocks organized in the same data page lead to accessing the same counter address. Therefore, most of the workloads tend to have high cache hit ratio of metadata. Typically, the split-counter organization can fit 64 data cache blocks' counters in a 64-B metadata cache block.

The frequently accessed and verified tree nodes in the MT can be cached on-chip as well [15, 49, 67]. This allows the integrity verification of a data block to complete as soon as the needed tree node is found in the on-chip cache. The reason being since the cached tree nodes have already been verified and their security is guaranteed being on-chip, it can be trusted as if it were the root of the tree. The tree nodes in higher layers cover larger range of memory data and show higher locality. The tree root protects the whole memory. In this article, we prefer to use 128-bit HMACs for all memory blocks, which have stronger security of memory integrity verification. Thus, one cache block can contain four HMACs.

2.5 NVM Crash Consistency

Programs can directly manipulate persistent data in-place in NVM with direct read and write accesses (e.g., load/store instructions) without using the conventional file system indirection for better performance. The persistent data maintained by the program is expected to be recovered to a consistent state in the event of a failure. Ensuring the recoverability of persistent data in a consistent state is referred to as the crash consistency guarantee. However, performance optimization techniques such as caching and write-back mechanisms in modern processors coalesce and reorder writes to NVM. Therefore, the writes that reach NVM can be different from the program order, which can lead to an unrecoverable state in the presence of system failures, such as power outages and application/kernel crashes. Take inserting a new node to the head of a persistent linked list in NVM as an example. The general insertion process consists of two steps: adding the new node to the head of the list and then making the head pointer point to the new node. After two steps finish, there exists a state when the valid data of the new node still exists in cache while the updated head pointer has been persisted in memory. The linked list can fail to recover to a consistent state if a power failure happens at that time, because after reboot, the head pointer points to an invalid place.

The x86-64 architecture specification now includes the low-level primitives (e.g., clflushopt, clwb, sfence [2]) that flush or write back a specific line from the cache hierarchy to memory[1] and enforce the ordering between writes. The sfence instruction stalls the thread until all of its outstanding instructions are finished before carrying out the next step. The instruction sequence is in the form of the following: clwb X; sfence;. It guarantees that when the sfence completes, the data at address X will survive a crash. Programmers use these operations directly by moving data to persistent memory for durability and ordering updates for consistency as needed, but it leaves the write latency on the critical path.

Prior works have proposed various crash consistency guarantees for NVM systems, including software-based solutions such as the log-based mechanism [14, 19, 20, 22, 27, 36, 37, 62, 64], shadow

[1]Intel proposed the PCOMMIT instruction to flush data from memory controller buffers but has deprecated it. Intel now requires platform support to flush memory controller buffers on a power failure [39].

paging [13, 43], language-level consistency [26], and other relaxed mechanisms [4, 16], as well as hardware-based solutions such as instruction set architecture extension [23, 28, 40, 45], transparent checkpointing [48], hardware logging [24, 25, 53], and other hardware modification [42, 69].

3 MOTIVATION

3.1 Crash Consistency Cost in Secure NVMs

Although placing the frequently accessed secure metadata into the on-chip cache improves performance, it causes consistency problems for NVMs. Once a system/power failure occurs, the cached metadata may lose, whereas its associated data might have already reached NVM. Such inconsistency between data and metadata can prevent us from correctly decrypting and authenticating the NVM data. In addition, the failure will cause lots of mismatch between the parent nodes and the child nodes, thus making plenty of data "corrupted" although they are correct. To guarantee that we can always use the correct data at runtime and retain secure protection after crashes, we need to ensure the following:

(1) The data block, associated counter, and data HMAC should reach memory atomically. Otherwise, the data cannot be decrypted or authenticated correctly.
(2) All layers of the MT in NVM along with the root node in the TCB should be consistently updated.

Specifically, updating of the MT is costly. Updates should be applied from the leaf node up to the root node (12 layers for a 16-GB NVM with 128-bit HMAC), which causes performance degradation, additional NVM write traffic, and extra security engine overhead.

Performance degradation. The calculation of each HMAC in the tree nodes must be executed one after another (instead of in parallel), since the parent node stores several counter HMACs, and each of them is generated with one of its child nodes. Thus, the write-back should wait until the tree root is updated, which significantly increases the latency of write operations. Furthermore, as described in Section 2.5, programs that need to ensure the crash consistency at the memory level are different from the conventional programs. For conventional software, memory reads issued by programs are on the critical path, whereas memory writes may be buffered, coalesced, and reordered on the way to memory for better performance. However, in crash-consistent software, these optimization on memory write might be broken because the order of writes to memory is severely constrained to ensure data recoverability across failures. These types of software use cache flush instructions to ensure the durability of writes, which places writes to memory on the critical path of program execution [35]. Due to the additional constraints on maintaining persistent security metadata in NVMs (e.g., MT updating), those on-the-critical-path writes further degrade performance due to the increased latency.

NVM write traffic. Since a write-back will incur the updates of the corresponding branch of the MT nodes, these updated nodes cause extra writes to NVM due to the crash consistency guarantee. Additional writes considerably increase the number of writes to NVM but meanwhile decrease the lifetime of the NVM. In addition, since writes to NVM consume more energy than writes to DRAM, the additional NVM writes also increase energy consumption of secure NVM systems.

Security engine overhead. Every time a data cache line is evicted from the last-level cache, it will incur the updating of the corresponding counters and the cascading computation of the MT nodes that lie on the branch of that counter. Compared to the scheme that stops calculating HMAC at the first cached tree nodes, crash consistency guarantee leads to lots of additional hash computation overhead, thus increasing energy consumption of the security engine.

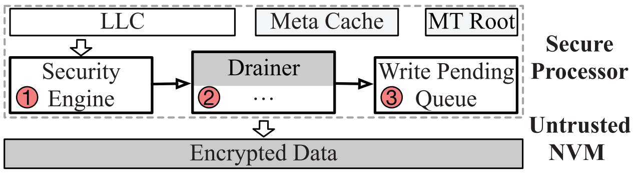

Fig. 6. Architecture of ShieldNVM.

These restrictions significantly limit the efficiency of secure NVM. We have implemented a naive approach (called *SC* in Section 5) to guarantee consistency, which ensures atomicity by aggressively flushing all of the related security metadata. Our evaluation shows that it can increase memory writes by 5.5x, deteriorate system performance by 62.8%, and increase the HMAC computation overhead by 8.3x over NVM workloads when compared to conventional security architecture without crash consistency guarantees.

3.2 Recovery Time for Secure Metadata

The system recoverability is the essential property that the crash-consistent software needs to be ensured. The encryption and integrity-protection mechanisms employed in the NVM systems should also consider the recoverability. The counters and the whole MT need to be recovered first to be able to decrypt and verify integrity before the programs continue execution. The recovery process is as follows: at first, recover the counters to their newest version, then rebuild the MT, which includes recursively calculating HMACs over the tree from the leaves to the root. Such a recovery process would take a lot of time for practical NVM capacities (e.g., 6 TB) and is considered unacceptable for applications that need high availability—for example, using the state-of-the-art counter recovery scheme [67], which relies on ECC bits of the data blocks as a sanity check for recovering the stale encryption counter. It needs to scan the whole memory for recovering. We calculate the recovery time by counting the number of data blocks, counters, and HMACs that need to be fetched and updated from memory, and assume each would cost 100 ns (fetch from memory, counter verification, or HMACs calculation, as assumed in Ye [67]). The recovery time can reach about 2.9 hours for 6-TB NVM capacity. Since the recovery time scales linearly with the size of the memory (number of data blocks), with the expected huge capacity of NVM, the recovery time for secure NVM systems is unacceptable. Anubis [70] addresses the recovery time of Osiris by tracking the addresses of metadata caches blocks in NVM. However, it needs to reserve spaces in NVM for tracking and protect the integrity of these spaces. In addition, the overhead of updating up to the root and single point of failure still exists.

4 DESIGN

4.1 Overview

Figure 6 shows the overall architecture of ShieldNVM. To improve performance, both the encryption counters and tree nodes of the MT (i.e., counter HMACs) are cached in the metadata cache (which is Meta Cache in Figure 6). For simplicity, we represent a counter cache and an MT cache with *Meta Cache* in the figure. The BMT root is stored in a persistent register, so as to prevent replay attacks after reboot. The key insight behind the ShieldNVM design lies in how to exploit the benefit of metadata caching to achieve efficient crash consistency guarantee while providing fast recovery and fine-grained data protection even after system crashes. For this purpose, ShieldNVM incorporates three components working together in the memory controller: *Security Engine*, *Drainer*, and *WPQ*.

To ensure that the MT nodes in NVM are always consistent with each other (i.e., the parent nodes should contain the correct hashes of the child nodes if there are no attacks happening), the atomic state transition of the BMT from the cache to NVM needs to be achieved. We first use an epoch-based mechanism to guarantee the consistency of the BMT when exploiting the benefits of metadata caching (see Section 4.2.1). For normal write-back data blocks (e.g., cache-line eviction and cache-line flushing), the *Encryption Engine* updates the corresponding security metadata directly in *Meta Cache*. Meanwhile, the addresses of all dirty cache lines in *Meta Cache* are persistently recorded by *Drainer* to speed up BMT state transition. Those cache lines corresponding to the recorded addresses need to be written back (see Section 4.2.2) when a draining event is triggered (see Section 4.2.3). In addition, since those addresses are persistently recorded, the stale secure metadata in memory can be identified by *Drainer* after crashes. Those memory blocks of secure metadata that are not recorded in *Drainer* are consistent with the blocks in the cache. At the recovery process, these recorded blocks need to be recovered (see Section 4.4).

Once a draining event is triggered, *Drainer* then atomically commits the updates in the current epoch to the *WPQ* (see Section 4.2.2), and finally, writes in the *WPQ* are guaranteed to reach NVM with the support of asynchronous DRAM refresh (ADR) [39]. Therefore, the MT in NVM is guaranteed to always be consistent, so its ability to specify the replay attacks still holds. In addition, by aggressively caching the metadata in *Meta Cache* and lazily evicting them out, write traffic to NVM is dramatically reduced. The duration between two adjacent committing points is known to be an epoch. After the draining events finish, the secure metadata blocks in NVM is up-to-date (consistent with the data in *Meta Cache*). Therefore, when crashes happen, the recovery process only need to recover the updates that are lost during one epoch. As well, the recovery time is significantly reduced due to the limited epoch length (see Section 4.2.3).

To reduce the overhead of calculating the counter HMACs for each data write-back event, the *Encryption Engine* stops calculating the counter HMACs for a tree node once its child has already been cached in *Meta Cache*. Instead, the update is spread to the root node only at the draining phase (see Section 4.3). Thus, the counter HMAC computation overhead for each epoch is reduced. In addition, the latency of the write-back event is decreased, which further reduces the crash consistency overhead.

After a system failure, however, a consistent but "old" MT in NVM may mismatch with the newest data blocks and data HMACs. As a result, the subsequent process of data decryption and integrity checking may fail. However, the data HMACs can be leveraged to recover those stale counters to their newest versions (see Section 4.4). The creation of data HMACs is to detect whether the value the processor write to the memory latest is the same as the value it load from that location. Any spoofing and splicing attack will cause the mismatch between the data and its HMAC. The sudden system failure might render the security metadata and program data partially inconsistent in the main memory, which also causes the mismatch between the data and its HMAC. Hence, a crash can be considered a type of attack that tampers the data. Data HMACs can act as a detector to pick out these stale counters caused by system crashes. Following this, we can further rebuild the MT with the newest counters and finally detect/locate possible attacks. However, scanning the whole memory for recovering the stale counters may take lots of time. We find that the persistent records in *Drainer* can be used to speed up the recovery process. The speedup results are shown in Section 6.7.

4.2 Epoch-Based Consistent BMT

The organization of the security metadata make them own high locality than memory data as shown in Section 2.4. To better utilize the locality of metadata while being able to minimize the crash consistency overhead, we propose *epoch-based consistent BMT*. It achieves this by aggressively caching the frequently accessed security metadata in *Meta Cache* and atomically committing these updates to the persistent domain.

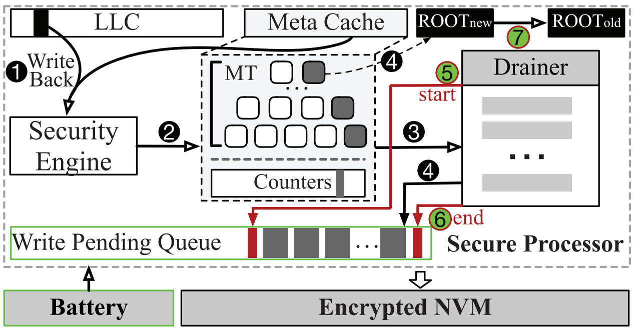

Fig. 7. Epoch-based consistent BMT.

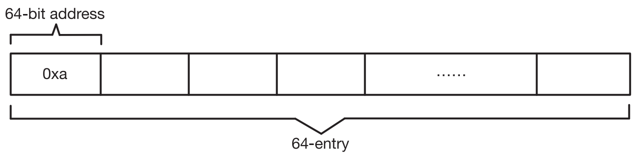

Fig. 8. *Dirty address queue* in *Drainer*.

4.2.1 Aggressive Caching. For normal write-back events, as shown in Figure 7, each time a data block is evicted or flushed from the last-level cache (in ❶), it is encrypted and authenticated by the *Encryption Engine* (in ❷). In this step, all metadata updates (e.g., adding 1 to the corresponding counter, modifying the related branch of tree nodes in the BMT from the counter's parent node to the tree root's child) are conducted directly in *Meta Cache*. In the meantime, *Drainer* tracks all of the dirty cache lines in *Meta Cache* by appending the related addresses of them into its *dirty address queue* (in ❸). Note that we skip those dirty cache lines if their addresses have already been put in the *dirty address queue*. Finally, the root node in the TCB is updated, and the write-back enters the WPQ (in ❹). The atomicity of root update and write-back is ensured using the persistent registers proposed in Awad et al. [6]. Actually, the process of ❷ and ❸ are executed in parallel. This is because for a specific data block, the related metadata addresses are deterministic in the existing secure NVM system. Typically, the time of updating the MT is longer than tracking addresses in the *dirty address queue*. Thus, the latency of tracking (❸) is hidden. Figure 8 shows how the *dirty address queue* is organized. Considering the 64-bit address, the storage overhead is minimal (e.g., since the size of the *dirty address queue* is limited by the entry number of the WPQ, for 64-entry WPQ and 1-TB NVM, the overhead is only 512 B/1 TB).

Although caching the metadata reduces write traffic to NVM, we still need to periodically commit the cached updates to NVM. There are mainly two challenges: (1) how to guarantee the atomicity of committing since the metadata in NVM needs to be consistently updated, and (2) how frequently we should commit (i.e., epoch length), which balances the gains of caching and the risk of data loss.

4.2.2 Atomic Draining. To address the first issue, we adopt the hardware ADR [39] mechanism. It ensures that any write requests buffered in the WPQ of the memory controller will be successfully written back to NVM with some backup power in case of a power failure. Thus, the number of entries in the WPQ limits the size of the persistent domains in processors. In our implemented system, we use a 64-entry (i.e., 4-kB) WPQ. We then add two persistent registers in the TCB. Among them, ROOT_{new} is updated for normal write-back events in step ❹, whereas ROOT_{old} is updated only at the committing phase.

Based on this, we propose the atomic draining protocol. To atomically commit the updates to NVM, *Drainer* first sends a *start* signal to the memory controller (step ❺) and writes back the dirty cache lines tracked in the *dirty address queue* to the memory controller. Once the memory controller receives the signal, it starts to block the metadata cache lines by putting them in the WPQ (normal data blocks still flow in legacy mode). When all of the related metadata cache lines have been sent to the memory controller, *Drainer* sends an *end* signal to notify the controller to flush all of the cache lines in the WPQ to NVM (step ❻). The dirty metadata cache lines are just written back but not invalidated. Finally, *Drainer* updates $\mathrm{ROOT}_{\mathrm{old}}$ with the value of $\mathrm{ROOT}_{\mathrm{new}}$, so $\mathrm{ROOT}_{\mathrm{old}}$ is consistent with the MT in NVM (step ❼). When a system failure occurs, the memory controller continues to flush the cache lines in the WPQ to NVM with the backup power provided by the ADR subsystem. However, if the system crashes before the memory controller receives the *end* signal, it just drops all residual cache lines in the queue, so as to keep the MT in NVM consistent. In addition, when the system crashes after the memory controller receives the *end* signal but before updating $\mathrm{ROOT}_{\mathrm{old}}$, we are sure that the newest consistent state of the MT will eventually reach NVM because of the ADR support. Thus, the MT in NVM is consistent with $\mathrm{ROOT}_{\mathrm{new}}$, and it still can conduct correct recovery and detect/locate attacks. In conclusion, we can ensure that the MT in NVM is always consistent with at least one of the roots in the TCB.

4.2.3 Epoch Length. To address the second issue, atomic draining is triggered when any one of the following events is satisfied:

(1) The *dirty address queue* is full, or it does not have enough entries to store the corresponding metadata addresses of the next evicted data block. In our implementation, the *dirty address queue* has the same number of entries with that in the WPQ.
(2) A cache line in *Meta Cache* is evicted by the cache system.
(3) A cache line in *Meta Cache* has been updated for more than N times since it becomes dirty (for fast recovery, in Section 4.4).

In addition, when a draining event is triggered, steps ❶ and ❷ for the subsequent evicted data blocks are blocked until the draining is finished. With such an epoch-based consistency mechanism, applications can fully exploit the benefits of metadata caching by maximizing the epoch length. Meanwhile, the MT in NVM is guaranteed to be consistently transferred from an old state to a new one.

4.3 Deferred Spreading

Considering the locality property of most workloads, it is very likely that several adjacent data blocks evicted by the CPU cache share the same ancestors of tree nodes in the MT. Therefore, updating the tree nodes for each data block independently can cause significant redundancy of computing (step ❷). The same tree nodes can be updated many times during one epoch. To eliminate such redundancy, we propose *deferred spreading* by calculating and updating the old tree nodes of the MT only at the draining phase.

4.3.1 Process with Deferred Spreading. The process of ShieldNVM with a deferred spreading mechanism is shown in Figure 9. Different from that depicted in Section 4.2, in step ❷, we stop updating the secure metadata when it has already been cached in *Meta Cache*. This is because the verified and cached tree nodes are always considered safe, which shares the same principles as that in a traditional DRAM-based secure system [15]. In step ❸, we still need to reserve entries in the *dirty address queue* for the related counter HMACs, despite the fact that they have not been updated yet. The reason we need to add those related addresses to the *dirty address queue* in advance is that we need to limit the total number of entries during the draining process below 64.

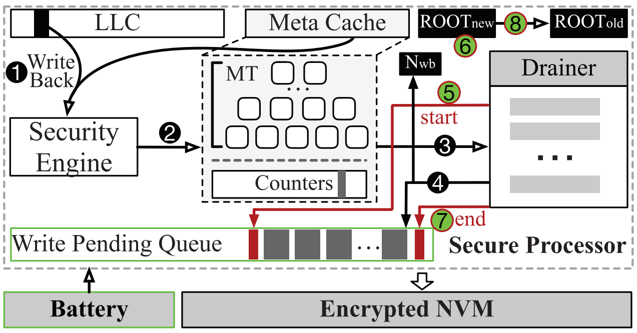

Fig. 9. ShieldNVM with deferred spreading.

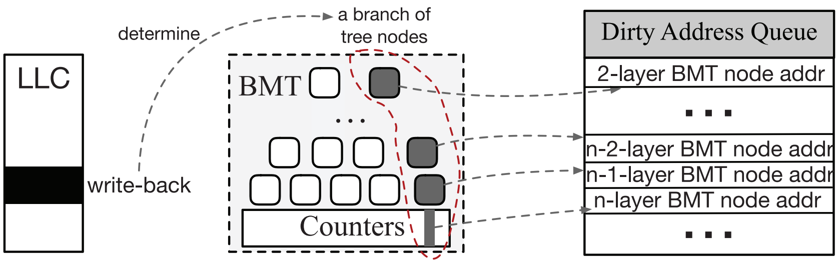

Fig. 10. Adding the addresses of a branch of tree nodes to the *dirty address queue*.

For a given address of the memory block, the addresses of its associated counter and a branch of the tree nodes are deterministic. Thus, the *dirty address queue* holds the invariant that at any time, it contains all dirty addresses (and those that will be updated in the draining phase) of the meta cache, and any clean addresses (which will not be updated at the draining phase) should not exist in it. The pre-add process is shown in Figure 10. The grey nodes are not updated immediately if the corresponding counter is found cached in *Meta Cache*. However, their addresses are also added to the *dirty address queue* with the counter's address. For the address that already existed in the *dirty address queue*, we just skip it.

During the draining phase, all tree nodes (i.e., counter HMACs) indexed by the *dirty address queue* are finally calculated and updated by the security engine, and ROOT_{new} in the TCB is updated accordingly (❻). Note that we still need two roots to keep that ROOT_{old} is consistent with the tree state of last epoch and ROOT_{new} is consistent with the current epoch. If the system crashes before the memory controller receives the *end* signal, it just drops all residual cache lines in the queue, so as to keep the MT in NVM consistent with ROOT_{old}. In addition, when the system crashes after the memory controller receives the *end* signal before updating ROOT_{old}, we are sure that the newest consistent state of the MT will eventually reach NVM because of the ADR support. Thus, the MT in NVM is consistent with ROOT_{new}. In conclusion, we can ensure that the MT in NVM is always consistent with at least one of the roots in the TCB. ShieldNVM with a deferred spreading mechanism still can conduct correct recovery and detect/locate attacks.

4.3.2 Potential Replay Attacks. However, introducing *deferred spreading* can lead to undetectable replay attacks. As shown in Figure 11, a system failure occurs during the epoch (i.e., after the last epoch is committed and before this epoch is committed). Thus, the MT along with the root node is still in an old state, despite that several data blocks have been newly written. If the newly written data and the associated data HMAC are replayed to their old version, such a replay attack

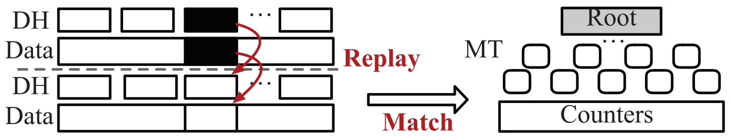

Fig. 11. Undetectable replay attack with an old root.

becomes undetectable because (1) the old MT in NVM is consistent with one of the roots in the TCB (both of them are lazily updated with the optimization in Section 4.3), and (2) the old counter still matches with the replayed data block and data HMAC. Actually, the key reason for such a potential replay attack is that the consistency between the data blocks and the MT is relaxed. By lazily updating ROOT_{new}, ROOT_{new} cannot reveal the most recent counter blocks, and the newly written data blocks are not always protected by an old MT that shows the last committed state. This kind of attack can only happen to the data that is written back in the latest epoch. To avoid such undetectable replay attacks, we add an extra 64-bit persistent register (a.k.a. N_{wb}) in the TCB, so as to record the number of write-back events since the last draining is committed. This record reveals how many times the counters have been updated since the last epoch. We use the persistent registers [6] to ensure the consistency between the N_{wb} and the write-back events.

In addition, all dirty addresses of the security metadata are persistently recorded in the *dirty address queue*. We can reduce the scope of this kind of replay attack to the data blocks corresponding to these addresses recorded by the *dirty address queue*. During recovery, we then use N_{wb} along with the recorded addresses to detect such replay attacks (see Section 4.4).

4.4 Crash Recovery

On a normal shutdown, ShieldNVM writes back all cache lines recorded in the *dirty address queue*, ensuring that the data in NVM is the newest and consistent. When the system crashes before a draining is committed, the MT in NVM may mismatch with the data blocks and data HMACs. Thus, we need a way to recover it to its newest version.

4.4.1 Utilizing the Data HMACs. The creation of data HMACs is to detect whether the value the processor writes to the memory latest is the same as the value it loads from that location. Any spoofing and splicing attack will cause the mismatch between the data and its data HMAC. The sudden system failure might render the security metadata and program data partially inconsistent in the main memory, which also causes the mismatch between the data and its data HMAC. In such a case, the HMAC calculated by the new data with an old counter does not equal the data HMAC stored in NVM. Hence, a crash can be considered a type of attack that tampers with the data. In addition, the inconsistency between the data and its associated counter can be discovered using the data HMACs.

Treating system failures as a type of attack and making full use of the data HMACs to detect a corrupted state achieves discovering the stale counters. However, the data HMACs can only achieve detecting old counters. Recovering the counter to its newest state can be achieved by utilizing the characteristic of the CME [67]. During runtime, a counter is incremented by one each time a data block is written back, so we can always recover a stale counter to its newest version by calculating the data HMAC with this counter and the corresponding data block, and compare it with the existing data HMAC in NVM as shown in Figure 12. If they are equal, this counter is consistent with the data block. Otherwise, we increase the counter by one and retry. To correctly work with such a recovery mechanism, the data block and data HMAC should be written back to NVM atomically for each write-back event. Luckily, this is not an issue since in existing BMT

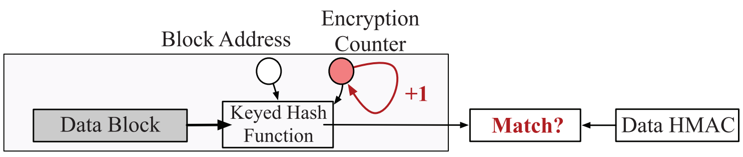

Fig. 12. Counter recovery process.

architecture, data HMACs are not cached in *Meta Cache* [49] and are generated directly in the memory controller. Therefore, such atomicity can be easily achieved with the existence of ADR.

4.4.2 Recovery Process. Since counters can be correctly recovered, the MT can be reconstructed as well. We divide the recovery process into three steps:

(1) *Recover the stall counters and locate attacks*: Once the system boots up, the system starts reading the content of *dirty address queue* to get the list of possibly lost updates in the cache. For each address, the data blocks (corresponding to the possibly lost counters) and its data HMACs are read and used to recover counters as discussed in Section 4.4.1. Meanwhile, we also count the times for recovering the stale counters.
(2) *Rebuild the MT based on the recovered counters*: Once the affected counters are fixed, ShieldNVM picks out the possibly lost updates in the first level (immediate parents of counters) from the *dirty address queue* and recalculate the nodes' values based on their immediate children (counters). Then, once the first level of the BMT is fixed, ShieldNVM searches for the second level and fixes them by calculating their values based on their immediate children in the first level. ShieldNVM proceeds with the same process by going up in the tree level by level and eventually reaches the root of the tree.
(3) *Detecting attacks coarsely*: For ShieldNVM without deferred spreading, once we reach the root level, the resulting root from the calculated tree will be compared against what is in the processor chip to find out if the recovery has been successful. If the resulting root mismatches with $\mathrm{ROOT}_{\mathrm{new}}$ in the processor chip, then the recovery process has failed, and we need a further procedure to find which part is attacked (see Section 4.4.3). For ShieldNVM, if the counted number is equal to N_{wb}, the recovery is considered successful. Then we update $\mathrm{ROOT}_{\mathrm{new}}$. Otherwise, we need a further procedure to find which part is attacked (see Section 4.4.3).

The security metadata recorded in the *dirty address queue* usually construct a sub-tree of the BMT. Thus, ShieldNVM only needs to read 64 addresses and their corresponding data blocks to construct the newest sub-tree instead of scanning the whole memory to reconstruct a whole BMT. Therefore, it can significantly speed up the recovery process. The pseudo-code of the recovery process is shown in Algorithm 1.

4.4.3 Detecting and Locating Process. Once the normal recovery procedure is not successful, we need to pick out which data has been tampered with and report it. To find out the tampered area, we need to scan the whole NVM. We divide the BMT into two parts according to the addresses tracked by *dirty address queue*. One part is the sub-tree that consists of memory blocks that are not updated during the latest epoch before crash. The addresses of the security metadata in this part cannot be in the *dirty address queue*. This part is the newest in NVM. We call this part *new sub-tree*. The other is the sub-tree that is constructed by the memory blocks that are updated during the latest epoch. The addresses of the security metadata in this part must be in the *dirty address queue*. The memory data in NVM is old since the newest data is in the cache and is lost. We call this part *old sub-tree*. Then we take steps to search the tampered area for the two parts respectively:

ALGORITHM 1: Recovery Process

```
1: Read *dirty address queue*;
2: **if** *dirty address queue* == $\phi$ **then**
3:     Finish Recovery;
4: **end if**
5: ct_queue ← corresponding counters recorded in the *dirty address queue*;
6: $N_{retry}$ ← 0;
7: **for** each $counter_i$ in ct_queue **do**
8:     **for** each $minor\_counter_j$ in $counter_i$ **do**
9:         Read corresponding $data\_blocks_j$ and $data\_HMAC_j$;
10:        Fixing the $minor\_counter_j$ using Counter Recovery Process illustrated in Section 4.4.1;
11:        added_times ← Totally added times for recovering the $minor\_counter_j$;
12:        $N_{retry}$ += added_times;
13:    **end for**
14: **end for**
15: dirty_node_queue ← ct_queue;
16: **while** dirty_node_queue ≠ $\phi$ **do**
17:    node ← the head of the dirty_node_queue;
18:    **if** node is the tree root **then**
19:        recovered_root ← node;
20:        Break;
21:    **end if**
22:    parent_node ← the node's parent;
23:    Calculate the new HMAC and put it into the parent node;
24:    **if** parent_node ∈ dirty_node_queue **then**
25:        Replace the existed parent_node;
26:    **else**
27:        Add the parent_node to the tail of the dirty_node_queue
28:    **end if**
29:    Remove node from the dirty_node_queue;
30: **end while**
31: **if** (ShieldNVM W/O Deferred Spreading) and (recovered_root == $ROOT_{new}$) **then**
32:    The system is successfully recovered to the newest;
33: **else if** ShieldNVM and ($N_{retry}$ == $N_{wb}$) **then**
34:    $ROOT_{new}$ ← recovered_root;
35:    The system is successfully recovered to the newest;
36: **else**
37:    Conduct further detecting illustrated in Section 4.4.3;
38: **end if**
```

(1) *Locate attacks happening on the new sub-tree*: With the epoch-based consistency mechanism, the BMT in NVM is ensured to be always consistent with at least one of the roots in the TCB in the absence of any attacks. Therefore, a replay attack can be located if any two of the parent and child tree nodes mismatch. We scan the whole NVM to check the data and its corresponding data HMAC, and the HMAC of the tree nodes with its parent. If any mismatch happens, we report the corresponding memory blocks. We do not conduct this checking process if ShieldNVM recovers successfully. This is because at runtime, MT integrity verification can pinpoint the tampered data in the new sub-tree.

Table 1. System Configuration

Processor	
CPU	Out-of-order cores x86-64, 3 GHz
L1 D/I cache	Private, 2 cycles, 32 KB, 2-way, 64-B block
L2 cache	Shared, 20 cycles, 256 KB, 8-way, 64-B block
PCM Main Memory	
PCM	16 GB, 60-ns read, 150-ns write [30, 66, 67, 70]
Memory controller	Data read/write queue: 32/64 entries
Dirty address queue	64 entries, 32 cycles (look-up latency)
Security Parameters	
AES (Encryption)	72 ns [57, 66]
SHA-1 (Integrity)	80 cycles [46, 49, 66], 128-bit HMAC
Counter cache	128 KB, 32 cycles, 8-way, 64-B block
MT Cache	128 KB, 32 cycles, 8-way, 64-B block
MT	12 levels, 4-ary, 64-B blocks on each level

(2) *Locate attacks happening on the old sub-tree*: When recovering the counters, we record the total number of retries (N_{retry}). If N_{wb} (see Section 4.3.2) and N_{retry} are not equal, we assert that an attack has occurred. By introducing an extra register of N_{wb}, we can now detect the potential replay attacks for the old sub-tree but still cannot locate the exact tampered data block. However, since we persistently record the dirty addresses of the modified security metadata in the latest epoch before the crash, the potentially attacked data is limited to these recorded addresses. We still cannot tell which data is attacked in those addresses, but we significantly reduce the damage caused by single-point failure. In addition, the probability of this attack occurring is extremely low (the *dirty address queue* can contain at most 42 dirty counters, whose size occupies only 0.0000156% of a 1-TB NVM). As well, such a replay attack is unable to be located precisely only when the system crashes, which further reduces the probability. Adding more bits in the *dirty address queue* to record the update times of each minor counter can help us precisely locate the tampered data blocks. Since we know the update times of the stale counters, we can directly add the stale counter with the recorded number (the update times), obtaining the newest counter. After recovering the stale counters to their newest, we compute the data HMAC using the data block and the recovered counter, and compare it with the HMAC stored in NVM. If they are not equal, we assert that the data block has been tampered with.

5 METHODOLOGY

In this section, we first describe the evaluation methodology and provide a short description of the evaluated designs, and present the workload we use in our experiment.

5.1 System Configuration

We model the hardware design in the cycle-accurate simulator Gem5 [8]. The system configuration is shown in Table 1. The simulated system consists of x86-64 out-of-order processors running at 3 GHz. All caches have 64-B blocks and use an LRU replacement policy. Without loss of generality, we model PCM technologies with read/write latency of 60 ns/150 ns [30] and 16-GB capacity, similar to related work [66, 67, 70]. The counter cache in our implementation is 128 KB with a 64-byte cache block. As each counter is 7 bits and 64 minor counters share a 64-bit major counter, a 128-KB counter cache can store 128-K counters. The memory system is backed by Intel's ADR [39, 50] support where all write requests in the WPQ can finally reach NVM in case of a failure. The *dirty*

Table 2. Evaluated Workloads

Workload	Description
Array Swap	Swap random items in a persistent array.
B-Tree	Insert random values into a persistent B-tree.
Hash Table	Insert random values to a persistent hash table.
Queue	Randomly en/dequeue items to/from a persistent queue.
Red-Black Tree	Insert random values into a persistent red-black tree.

address queue in our implementation has 64 entries, which is equal to the number of entries in the WPQ. In Section 6.5, we show the effect of different number of entries. We store the root of the MT in a non-volatile register in the secure processor, as proposed in previous work [35, 66, 67].

We evaluate our scheme based on the baseline system that we call *w/o crash consistency* (w/o CC) and several state-of-the-art schemes. Here are the different designs we use in our evaluation:

—*W/o crash consistency (w/o CC)* is secure NVM without crash consistency. It only writes to memory dirty evictions from the cache. This is our normalized base.
—*Strict consistency (SC)* enforces the data block and the corresponding metadata (e.g., the counter and tree nodes in MT) to be atomically written to NVM, with root consistently updated in the TCB. The atomic mechanism is based on the persistent registers [67].
—*Osiris Plus* is an optimized version of Osiris that eliminates the need for evicting dirty counter blocks at the cost of extra online checking to recover the most recent counter value [67].
—*ShieldNVM w/o DS* is our base solution without deferred spreading (DS). It eliminates the need for atomic updates for every data block while minimizing the performance and write traffic overheads.
—*ShieldNVM* is an optimized version that only updates up to the cached MT node instead of up to the root. The update times are limited to 16 for Osiris Plus, ShieldNVM w/o DS, and ShieldNVM. In addition, we use a 64-entry *dirty address queue* for ShieldNVM w/o DS and ShieldNVM.

5.2 Workloads

Our evaluation uses five NVM workloads (listed in Table 2) that manipulate different persistent data structures. These workloads are similar to those used in prior works on persistent memory systems [12, 16, 34, 35, 48].

These workloads need persistency support that includes the clwb instruction, which writes back cache lines, and sfence, which ensures that any store instruction preceding the sfence instruction in the program order completes before any store instruction that comes after sfence. We instrument our workloads with clwb and sfence instructions in the appropriate places. The workloads are single threaded unless explicitly mentioned.

6 EVALUATION

First, we evaluate the impact of different designs (listed in Section 5.1) on system performance. Then, we compare the write traffic and HMAC computation overhead in these designs. Next, we evaluate the sensitivity of the results when we vary different parameters. Last, we compare the recovery time for different designs.

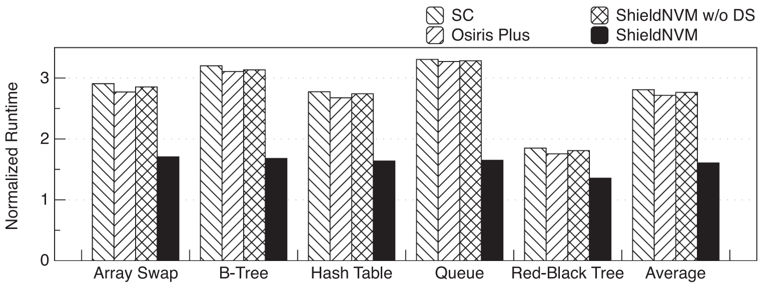

Fig. 13. System runtime for different designs. Results are normalized to w/o CC.

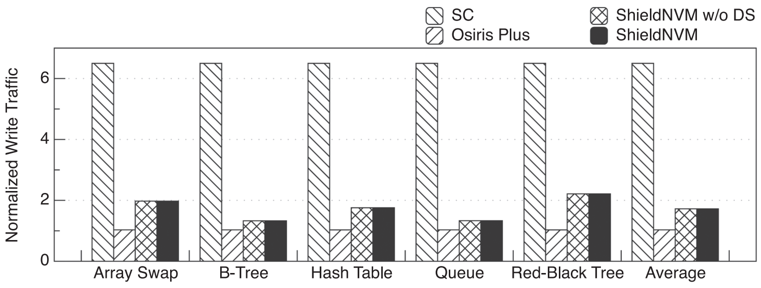

Fig. 14. NVM write traffic for different designs. Results are normalized to w/o CC.

6.1 System Performance

Figure 13 illustrates the impact of different designs on system performance in execution time over different benchmarks. We have the following observations. First, SC, Osiris Plus, and ShieldNVM w/o DS show very close but lower system performance compared to ShieldNVM. The clwb and sfence instructions are used to guarantee the persistence of the data in these workloads, which puts write operations on the critical path of the program execution. In addition, to guarantee the correct recovery, the write-back data need to ensure the consistent updates between the write-back data and the root node in the MT. Therefore, only when the root node is updated can the data blocks be forwarded to the WPQ, which significantly increases the write latency. Thus, SC, Osiris Plus, and ShieldNVM w/o DS slow down system performance significantly compared to w/o CC. They increase the runtime over NVM workloads, on average 181%, 171%, and 176%, respectively. Instead, for the ShieldNVM scheme, write-back data can be forwarded to the WPQ as long as an accessed metadata is cached in the TCB. Thus, compared to the other three schemes, ShieldNVM achieves lower write latency and better performance. Second, we observe that Osiris Plus performs slightly better than ShieldNVM w/o DS. This is because Osiris Plus eliminates the write traffic for MT node updates. Third, the performance of ShieldNVM is still lower than the baseline. In ShieldNVM, all write-back data needs to wait until all corresponding secure metadata addresses are put into the *dirty address queue*. The write latency is increased due to those search and put operations for the *dirty address queue*. Specifically, it reduces the system runtime by 41.3%, 39.1%, and 40.3% on average over SC, Osiris, and ShieldNVM w/o DS, respectively.

6.2 NVM Write Traffic

Figure 14 shows the comparison of memory write traffic incurred by different designs over the baseline w/o CC. Strict consistency has the most number of writes. It increases the number of writes by 5.5x. Since a BMT is a multi-level tree of hashes with counters as its leaf nodes, in strict

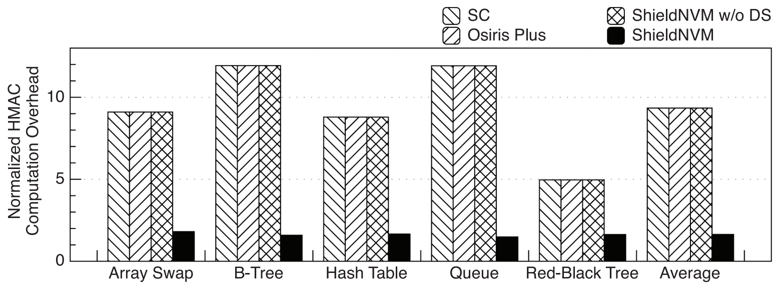

Fig. 15. HMAC computation overhead for different designs. Results are normalized to w/o CC.

consistency, all BMT nodes that lie on the branch of the modified counter on every write-back need to be written back. In our simulation, a 16-GB NVM with a 12-level 4-ary BMT requires 12 atomic BMT updates on every write-back (the BMT root is updated on the TCB, whereas 10 internal path nodes and the leaf-level counter are updated in NVM). This results in high memory write traffic, which negatively impacts the NVM lifetime. Both ShieldNVM and ShieldNVM w/o DS shows similar memory write traffic. This is because the deferred spreading mechanism has little impact on the *Epoch Length*. Thus, the number of draining events is almost the same for ShieldNVM and ShieldNVM w/o DS. Specifically, their write traffic is 72% higher than the baseline. The extra write traffic is introduced when the updated tree nodes in the MT is flushed to NVM. Osiris Plus has less write traffic than ShieldNVM. This is because it does not have to persist the tree nodes of the MT. However, we notice that the performance of ShieldNVM is higher than Osiris Plus, despite the fact that ShieldNVM incurs higher write traffic. This is because (1) the longer write latency caused by updating up to the root plays a major role in affecting the system performance, and (2) all extra metadata write traffic is incurred by data write-back, and they can reach WPQ quickly from *Meta Cache* in the memory controller. In addition, the NVM bandwidth is not the bottleneck in our tests.

6.3 Security Engine Overhead

Figure 15 shows the comparison of HMAC computation times incurred by different designs over the baseline w/o CC. Strict consistency, Osiris Plus, and ShieldNVM w/o DS have the highest number of HMAC computation times. These schemes increase the HMAC computation times by 8.34x on average. Since all of these three schemes need to update the HMAC from the leaf to the root on every write-back data, the increased HMAC computation times are the same. ShieldNVM incurs minimal additional HMAC computation overhead because of the deferred spreading mechanism. For normal write-back data, it only updates up to the first cached node. Only when the draining event is triggered does ShieldNVM generate extra HMAC computation. Because of the great locality of the security metadata, ShieldNVM only incurs an additional 63% of HMAC computation overhead.

6.4 Sensitivity to Cache-Line Update Times

There are two parameters we can change to affect the epoch length. One is the number of entries (M) in the *dirty address queue*, and the other is the limit of update times (N) for the dirty cache blocks in the metadata cache. In this experiment, we evaluate the overhead of ShieldNVM with variable N. The runtime overhead is shown in Figure 16. Table 3 shows the number of draining processes triggered by two conditions in the experiment. Since the cache-line eviction in *Meta Cache* hardly happens in our experiment, we did not count this data in the table.

Figure 16(a) compares the performance of ShieldNVM and ShieldNVM w/o DS when varying the N from 4 to 64. The *y*-axis shows the runtime normalized to the corresponding ShieldNVM

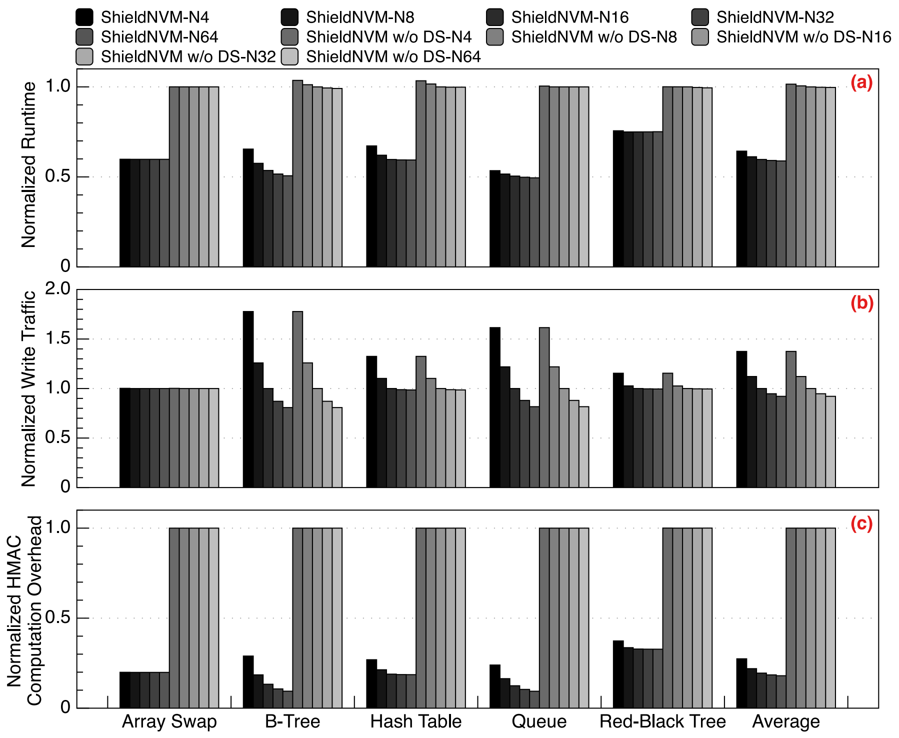

Fig. 16. Evaluating ShieldNVM and ShieldNVM w/o DS with different cache-line update times limit (N). The results are normalized to ShieldNVM w/o DS that has an update times limit set to 16 (ShieldNVM w/o DS-N16). All designs use a 64-entry *dirty address queue*. (a) Normalized runtime. (b) Normalized write traffic. (c) Normalized HMAC computation overhead.

Table 3. Atomic Draining Frequency Triggered by Two Events

	Array Swap		B-Tree		Hash Table		Queue		Red-Black Tree	
	NN	FULL	NN	FULL	NN	FULL	NN	FULL	NN	FULL
ShieldNVM-N4	265	5,771	135,390	0	15,010	0	25,027	0	24,895	684
ShieldNVM-N8	0	5,974	66,492	0	7,369	132	12,499	0	6,930	8,483
ShieldNVM-N16	0	5,974	32,117	0	921	3,867	6,249	0	530	13,036
ShieldNVM-N32	0	5,974	15,042	0	139	4,378	3,124	0	156	13,209
ShieldNVM-N64	0	5,974	6,682	0	35	4,425	1,562	0	63	13,230

One event is that a cache line in *Meta Cache* has been updated more than N times (NN). The other event is that the *dirty address queue* is full (FULL).

w/o DS that has the default parameter (ShieldNVM w/o DS with N set to 16 and M set to 64). We observe that when the update times limit is small (N is set to 4), the decreased runtime of ShieldNVM is 35.7% and the increased runtime of ShieldNVM w/o DS is 1.5% on average. The overhead decreases as the limit of update times increases. When N is set to 64, ShieldNVM can achieve a declining runtime by 41.2% and ShieldNVM w/o DS has a slight decrease of about 1% on average. As the N becomes larger, the epoch length becomes longer on average. Thus, the number of triggered draining processes decreases. Since the old HMACs of the BMT are updated at the draining process, the performance of ShieldNVM is more sensitive to the times of the draining events. However, this update times limit has slight impact on the performance of ShieldNVM w/o

Table 4. Atomic Draining Frequency Triggered by Two Events

	Array Swap		B-Tree		Hash Table		Queue		Red-Black Tree	
	NN	FULL	NN	FULL	NN	FULL	NN	FULL	NN	FULL
ShieldNVM-M32	0	20,270	32,117	0	68	16,646	6,249	0	183	40,776
ShieldNVM-M40	0	13,890	32,117	0	174	10,693	6,249	0	275	28,158
ShieldNVM-M48	0	10,110	32,117	0	292	7,484	6,249	0	320	20,867
ShieldNVM-M56	0	7,666	32,117	0	505	5,430	6,249	0	396	16,238
ShieldNVM-M64	0	5,974	32,117	0	921	3,867	6,249	0	530	13,036

One event is that a cache line in *Meta Cache* has been updated more than N times (NN). The other event is that the *dirty address queue* is full (FULL).

DS. The reason is that there is no need for ShieldNVM w/o DS to update the old BMT nodes since all nodes are new in the meta cache.

Figure 16(b) compares the write traffic of ShieldNVM and ShieldNVM w/o DS. A larger update times limit results in a longer epoch, which also reduces the number of writes to NVM. For both ShieldNVM-N64 and ShieldNVM w/o DS-N64, they reduce write traffic by 8% on average compared to ShieldNVM w/o DS-N16. A smaller limit results in more draining processes, increasing the writes to NVM. The reason is that when draining events are triggered, all corresponding tree nodes need to be flushed to NVM. For both ShieldNVM-N4 and ShieldNVM w/o DS-N4, they increase write traffic by 37.4% on average compared to ShieldNVM w/o DS-N16.

Figure 16(c) compares the HMAC computation overhead of ShieldNVM and ShieldNVM w/o DS. For ShieldNVM, lots of HMAC computation happen at the draining process. With a longer epoch, HMAC computation overhead can be declined due to the reduced draining times. On average, ShieldNVM-N64 reduces about 34.4% HMAC computation overhead compared to ShieldNVM-N4 and 82% compared to ShieldNVM w/o DS-N16. For ShieldNVM w/o DS, the HMAC computation overhead keeps constant for varying N. The reason is that no extra HMAC computation needs to be conducted at the draining process. Thus, elongating the epoch length does not affect the HMAC computation overhead for ShieldNVM w/o DS.

Notably, for the Array Swap workload, we observe that the performance, write traffic, and HMAC computation overhead keep constant with varying N. The reason is that changing the update times limit does not change the number of draining times for Array Swap in the experiment, as shown in Table 3. The limitation of entry number in the *dirty address queue* is dominant for the times of atomic draining. For the Hash Table workload, when the N is larger than 16, the effect of N slows down. This is because the M trigger condition plays major roles.

6.5 Sensitivity to Dirty Address Queue Size

In this experiment, we compare the performance (Figure 17(a)), write traffic (Figure 17(b)), and security engine overhead (Figure 17(c)) of ShieldNVM and ShieldNVM w/o DS with varying *dirty address queue* entries. Table 4 shows the number of draining processes triggered by two conditions in the experiment. Since the cache-line eviction in *Meta Cache* hardly happens in our experiment, we did not count this data in the table.

Figure 17(a) compares the performance of ShieldNVM and ShieldNVM w/o DS when varying the M from 32 to 64. The y-axis shows the runtime normalized to the corresponding ShieldNVM w/o DS that has the default parameter (ShieldNVM w/o DS with N set to 16 and M set to 64). We observe that with a smaller size (e.g., 32-entry *dirty address queue*), the decreased runtime of ShieldNVM is 35.7% and the increased runtime of ShieldNVM w/o DS is 1.5% compared to ShieldNVM w/o DS-M64 on average. For self-comparison, ShieldNVM-M64 increases performance by 7.2% compared to ShieldNVM w/o DS-M32. As the M becomes larger, the *dirty address queue* can hold more records

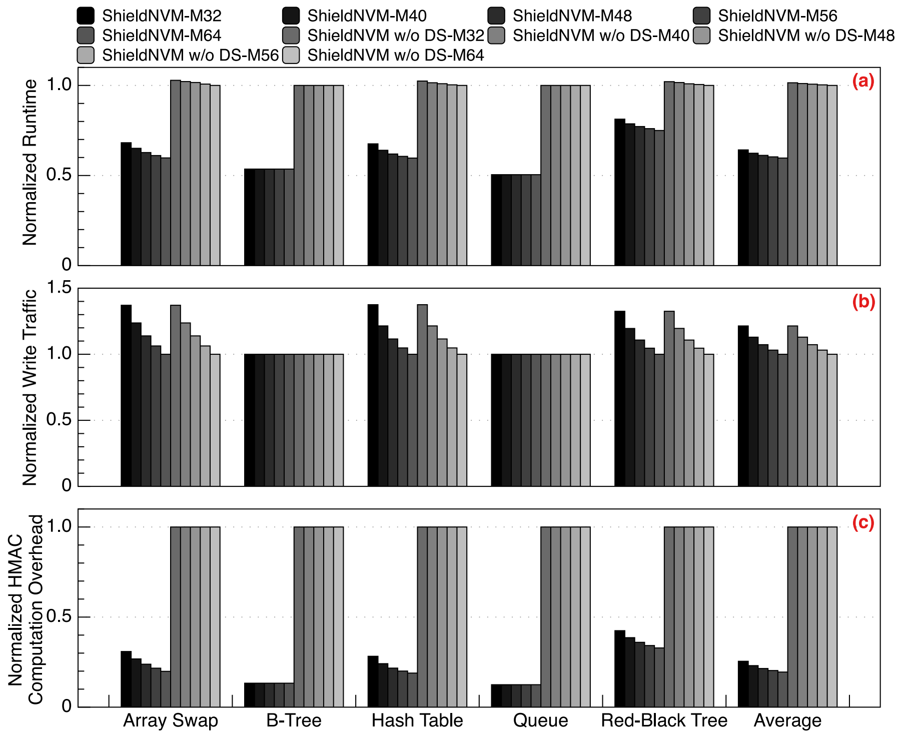

Fig. 17. Evaluating ShieldNVM and ShieldNVM w/o DS with different numbers of entries in the *dirty address queue* (M). The results are normalized to ShieldNVM w/o DS that has a 64-entry *dirty address queue* (ShieldNVM w/o DS-M64). All designs use a 16 update times limit. (a) Normalized runtime. (b) Normalized write traffic. (c) Normalized HMAC computation overhead.

before the draining point, elongating the epoch length on average. As a consequence, for a specified workload such as Array Swap, Hash Table, and Red-Black Tree shown in Table 4, the number of triggered draining processes decreases. As described in Section 6.4, the performance of ShieldNVM w/o DS is less sensitive to the frequency of the draining events. For ShieldNVM w/o DS, the HMAC computation latency is added to every write-back data, reducing the overhead at the draining process. However, for ShieldNVM, although many HMAC computations need to be considered during the draining process, lots of redundant HMAC computation overhead can be eliminated for normal operations (during the epoch), thus significantly increasing system performance.

Figure 17(b) shows the impact of different M on write traffic of ShieldNVM and ShieldNVM w/o DS. ShieldNVM-M32 adds an extra 21.4% of write traffic compared to ShieldNVM-M64. Since ShieldNVM and ShieldNVM w/o DS have almost the same number of draining events, they show the same tendency on write traffic with varying numbers of entries in the *dirty address queue*.

Figure 17(c) demonstrates HMAC computation overhead for different workloads with varying numbers of entries in the *dirty address queue*. On average, ShieldNVM-M64 reduces HMAC computation overhead by 18.2% compared to ShieldNVM-M32 and 80.5% compared to ShieldNVM w/o DS-M64. For ShieldNVM w/o DS, the HMAC computation overhead remains stable for different numbers of entries in the *dirty address queue*. Since HMAC computation is conducted from the leaf up to the root for every normal write-back operations, at the draining process, no old tree node needs to be updated. Thus, no additional HMAC computation happens at the draining process.

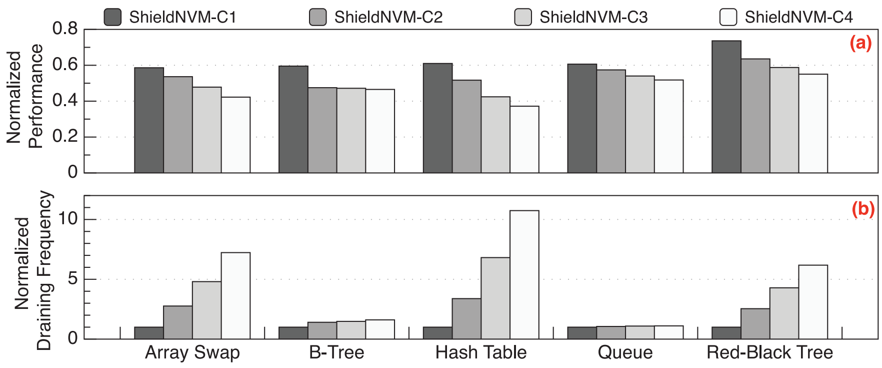

Fig. 18. Evaluating ShieldNVM in multi-core applications. All designs use a 16 update times limit and a 64-entry *dirty address queue*. (a) Results are normalized to w/o CC with corresponding core numbers. (b) Results are normalized to ShieldNVM-C1 (one core).

It is worth mentioning that for the B-Tree and Queue workloads, the performance, write traffic, and HMAC computation overhead remain constant for both ShieldNVM and ShieldNVM w/o DS. The reason is that for these two workloads in our experiment, the cache-line update times limit (N) is dominant for the epoch length. Varying M does not impact the times of the draining process, as shown in Table 4.

6.6 Multi-Core Performance

In this experiment, we compare the performance (Figure 18(a)) and draining frequency (Figure 18(b)) of ShieldNVM with varying core numbers. Each thread performs the same operations on different cores. As the number of cores increases, instead of keeping constant, the performance of ShieldNVM normalized to w/o CC with corresponding core numbers decreases, as shown in Figure 18(a). This is because the draining is triggered more frequently in multi-core workloads. As shown in Figure 18(b), the draining frequency increases with increased numbers of cores. The number of entries in the *dirty address queue* is only 64, and all cores share the same memory controller. The *dirty address queue* is more likely to be full under multi-core applications.

6.7 Recovery Time

To evaluate the recovery time, we calculate it by counting the number of data blocks and tree nodes that need to be fetched and updated from memory and assume that each would cost 100 ns (fetch from memory, hash calculation, and/or decryption, as assumed in Ye [67]). The counters' recovery needs additional checking. For counters that have mismatched checking, an average of eight trials (when N equals 16) of counter values will be tried before finding the correct value. This extra checking time will be added to the recovery time. We model and compare the following three schemes:

—*Osiris* is a stop-loss counter update mechanism without the need for evicting dirty counter blocks at the cost of extra online checking to recover the most recent counter value [67].
—*AGIT* is memory controller design based on Osiris that enables low recovery time for integrity-protected systems [70].
—*ShieldNVM* is an optimized version that only updates up to the cached MT node instead of up to the root.

Figure 19 shows the recovery time for different designs with varying NVM memory size. The recovery time of AGIT and ShieldNVM is not affected by the NVM capacity. Thus, it remains constant. For the 16-GB NVM size, ShieldNVM spends about 0.0022 seconds to recovery and AGIT

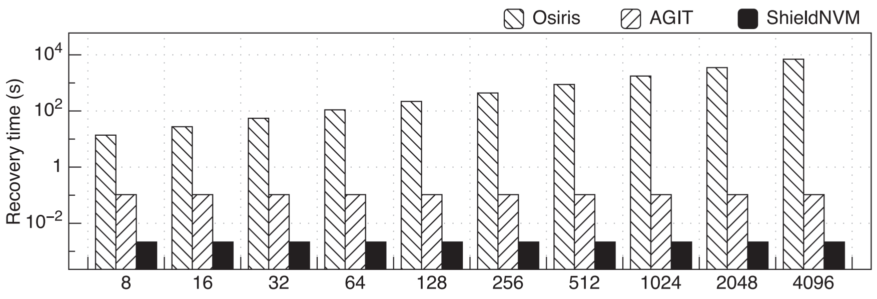

Fig. 19. Recovery time (logarithmic) with different NVM capacity (GB).

spends about 0.105 seconds. ShieldNVM is 48x faster than AGIT. For Osiris, it needs to spend 27.5 seconds, which is significantly slower than AGIT and ShieldNVM. With a larger NVM size, the difference increases. This is because for Osiris's recovery process, it first needs to recover the encryption counters. Since the counters are the leaves of the BMT and upper levels of BMT nodes are simply the hash values of lower levels, after recovering all stale counters, it then builds inner nodes of the BMT layer by layer. Osiris cannot identify which part of the counters are stale, and therefore it needs to scan the whole memory. The recovery time of Osiris is proportional to the NVM memory capacity.

AGIT is the state-of-the-art design that addresses the recovery speed of the secure NVM. It reduces the recovery time by tracking the addresses of the blocks in the MT and counter caches in the specified area called *shadow tracking* in NVM. Whenever metadata is first modified in the counter cache or MT cache, the corresponding address is also written to NVM. At the recovery process, AGIT first scans the content of the shadow-tracking area to get the possibly lost updates in the cache. For each address, the related 64 data blocks (corresponding to the possibly lost counters) are read and used to recover the counter using Osiris. Later, once the affected counters are fixed, AGIT scans through the shadow-tracking area to search for possibly lost updates in the first level (immediate parents of counters) and recalculates the nodes' values based on their immediate children (counters). AGIT proceeds with the same process by going up in the tree level by level and eventually reaching the root of the tree. Once at the root level, the resulting root from the calculated tree will be compared against what is in the processor chip to find out if the recovery has been successful. AGIT's recovery time is a function of the counter cache and MT cache size.

Compared to Osiris, ShieldNVM achieves significant speedup in recovery because ShieldNVM does not need to scan the whole memory during the recovery process. Compared to AGIT, ShieldNVM can still save recovery time because of our proposed epoch-based mechanism. In ShieldNVM, the number of stale metadata blocks is limited to be less than the number of entries in the *dirty address queue*. In addition, we only need to recover the blocks recorded by the *dirty address queue*. Thus, ShieldNVM's recovery time is a function of the *dirty address queue* size, which is smaller than the meta cache size.

7 RELATED WORK

In this section, we discuss relevant studies on memory security and NVM crash consistency.

Memory security. For DRAM security, several works mainly solve the performance issue. Vault [61] proposes efficient integrity verification structures to reduce paging overheads. Morphable counters [51] present a more compact integrity tree to enhance performance and reduce counter overflow. Synergy [52] aims to improve performance of secure execution while providing strong reliability for systems. It is based on the insight that MACs are capable of detecting data tampering and are also useful for detecting memory errors, which is similar to the observation that data HMACs can be utilized to detect inconsistency between the counter and data.

There are several state-of-the-art works on NVM security without considering crash consistency. DEUCE [68] and SECRET [59] focus on enhancing the lifetime of the encrypted NVM. ACME [57] and COVERT [56] mitigate the counter overflow in encrypted NVM. Silent Shredder [5] proposes to eliminate writes when writes are just zeroing out physical pages to achieve better power consumption and performance. i-NVMM [11] mainly solves en/decryption overhead during the normal execution time. Mao et al. [38] protect NVM from wear-out attack based on the timing difference of row buffer hit/miss. ASSURE [46] reduces the energy consumption of authenticated NVM.

Some works are aimed at bridging the gap between memory persistence and memory security. Selective persistence [34] proposes a counter-atomic mechanism by exploiting the application-level persistence semantics. It can selectively relax the atomicity between the data blocks and counters. SuperMem [72] proposes a cache write-through scheme to ensure the atomicity between the data blocks and counters. Although they provide data recoverability, they ignore the integrity protection of data and counters, which is essential for counter-mode memory encryption security [49]. ARSENAL [58] and Osiris [67] address the crash consistency issue in encrypted and authenticated NVM, but they do not solve the very long recovery time in secure NVM systems. Anubis [70] is based on Osiris and is devoted to achieving very low recovery time in secure NVM systems. However, recording metadata cache in NVM still incurs some overhead. In addition, ARSENAL, Osiris, and Anubis suffer from long write latency due to the HMAC computation from the leaf to the root. Triad-NVM [6] addresses recovery of general MTs on systems with both persistent and non-persistent regions, and is orthogonal to our work.

NVM crash consistency. The efficiency of crash consistency of persistent memory has been intensively studied in recent years. Prior works have proposed and implemented a variety of solutions to mitigate crash consistency overhead. We discuss them with two categories: reducing the persistence overhead and relaxing the ordering overhead. Reducing the persistence overhead includes introducing additional persistent part to the CPU chip [18, 22, 42, 69] or moving persistence out of the critical path [9, 33, 36, 48]. The most related work is LAD [18]. Both LAD and ShieldNVM use the persistent domain in the MC to achieve atomic durability. LAD mainly address the logging overhead in persistent transaction memory. ShieldNVM focuses on the atomic durability of security metadata.

Relaxing the ordering overhead includes hardware-based approaches, such as BPFS [13], DPO [28], HOPS [40], ATOM [25], LB++ [23], LOC [37], HPT [27], and different persistency modes [45]. With regard to software-based methods, these works utilize low-level primitives to provide high-level software interfaces (e.g., transaction memory, file system) for managing persistent data or designing data structures for persistent memory to eliminate ordering overhead. Transaction memory designs include Mnemosyne [62], NV-Heaps [12], LSNVMM [20], PMDK [3], Pisces [17], and OneFile [47], among others. There are also NVM-optimized file systems, such as PMFS [14], NOVA [64], HiNFS [10], and SCMFS [63]. Persistent memory data structure designs include CCEH [41], Level-Hash [71], FPTree [44], and FAST-FAIR [21], among others.

8 CONCLUSION

ShieldNVM achieves a low overhead crash consistency guarantee and employs a fast recoverable mechanism for encrypted and authenticated NVM. ShieldNVM implements epoch-based consistent BMT with deferred spreading to achieve high performance and write efficiency, as well as low security engine overhead. After crashes, ShieldNVM is capable of fast recovering security metadata, detecting/locating tampered areas, and continuing normal secure protection. In comparison to the state-of-the-art secure NVM Osiris [67] that guarantees crash consistency, ShieldNVM reduces system runtime by 39.1% and security engine overhead by 80.5% on average over NVM workloads. ShieldNVM also achieves fast recovery time for a secure NVM system that is much faster than Osiris and AGIT with varying NVM capacity. Once the normal recovery procedure is not successful, ShieldNVM holds the ability to locate tampered areas after crashes.

REFERENCES

[1] Intel. 2015. Intel and Micron Produce Breakthrough Memory Technology: 3D XPoint. Retrieved April 12, 2020 from https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/.

[2] Intel. 2018. Intel Architecture Instruction Set Extensions Programming Reference. https://software.intel.com/sites/default/ files/managed/c5/15/architecture-instruction-set-extensions-programming- reference.pdf.

[3] Pmem.io. 2019. Persistent Memory Development Kit. Retrieved April 12, 2020 from https://pmem.io/pmdk/.

[4] Mohammad Alshboul, James Tuck, and Yan Solihin. 2018. Lazy persistency: A high-performing and write-efficient software persistency technique. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA'18)*. IEEE, Los Alamitos, CA, 439–451. DOI: https://doi.org/10.1109/ISCA.2018.00044

[5] Amro Awad, Pratyusa Manadhata, Stuart Haber, Yan Solihin, and William Horne. 2016. Silent Shredder: Zero-cost shredding for secure non-volatile main memory controllers. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*. ACM, New York, NY, 263–276. DOI: https://doi.org/10.1145/2872362.2872377

[6] Amro Awad, Mao Ye, Yan Solihin, Laurent Njilla, and Kazi Abu Zubair. 2019. Triad-NVM: Persistency for integrity-protected and encrypted non-volatile memories. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA'19)*. ACM, New York, NY, 104–115. DOI: https://doi.org/10.1145/3307650.3322250

[7] Eduardo B. 2017. Enhancing High-Performance Computing with Persistent Memory Technology. Retrieved April 12, 2020 from https://software.intel.com/en-us/articles/enhancing-high-performance-computing-with-persistent-memory-technology.

[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, et al. 2011. The Gem5 simulator. *SIGARCH Computer Architecture News* 39, 2 (Aug. 2011), 1–7. DOI: https://doi.org/10.1145/2024716.2024718

[9] Sui Chen, Faen Zhang, Lei Liu, and Lu Peng. 2019. Efficient GPU NVRAM persistence with helper warps. In *Proceedings of the 56th Annual Design Automation Conference (DAC'19)*. ACM, New York, NY, Article 155, 6 pages. DOI: https://doi.org/10.1145/3316781.3317810

[10] Youmin Chen, Jiwu Shu, Jiaxin Ou, and Youyou Lu. 2018. HiNFS: A persistent memory file system with both buffering and direct-access. *ACM Transactions on Storage* 14, 1 (April 2018), Article 4, 30 pages. DOI: https://doi.org/10.1145/3204454

[11] S. Chhabra and Y. Solihin. 2011. i-NVMM: A secure non-volatile main memory system with incremental encryption. In *Proceedings of the 2011 38th Annual International Symposium on Computer Architecture (ISCA'11)*. 177–188. DOI: https://doi.org/10.1145/2000064.2000086

[12] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. ACM, New York, NY, 105–118. DOI: https://doi.org/10.1145/1950365.1950380

[13] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*. ACM, New York, NY, 133–146. DOI: https://doi.org/10.1145/1629575.1629589

[14] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys'14)*. ACM, New York, NY, Article 15, 15 pages. DOI: https://doi.org/10.1145/2592798.2592814

[15] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas. 2003. Caches and hash trees for efficient memory integrity verification. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA'03)*. 295–306. DOI: https://doi.org/10.1109/HPCA.2003.1183547

[16] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. 2018. Persistency for synchronization-free regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*. ACM, New York, NY, 46–61. DOI: https://doi.org/10.1145/3192366.3192367

[17] 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 *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'19)*. 913–928. https://www.usenix.org/conference/atc19/presentation/gu

[18] Siddharth Gupta, Alexandros Daglis, and Babak Falsafi. 2019. Distributed logless atomic durability with persistent memory. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*. ACM, New York, NY, 466–478. DOI: https://doi.org/10.1145/3352460.3358321

[19] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical persistence for multi-threaded applications. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys'17)*. ACM, New York, NY, 468–482. DOI: https://doi.org/10.1145/3064176.3064204

[20] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. 2017. Log-structured non-volatile main memory. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*. 703–717. https://www.usenix.org/conference/atc17/technical-sessions/presentation/hu.

[21] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable transient inconsistency in byte-addressable persistent B+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*. 187–200. https://www.usenix.org/conference/fast18/presentation/hwang.

[22] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-atomic persistent memory updates via JUSTDO logging. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*. ACM, New York, NY, 427–442. DOI: https://doi.org/10.1145/2872362.2872410

[23] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient persist barriers for multicores. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, 660–671. DOI: https://doi.org/10.1145/2830772.2830805

[24] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas. 2018. DHTM: Durable hardware transactional memory. In *Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. 452–465. DOI: https://doi.org/10.1109/ISCA.2018.00045

[25] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra. 2017. ATOM: Atomic durability in non-volatile memory through hardware logging. In *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*. 361–372. DOI: https://doi.org/10.1109/HPCA.2017.50

[26] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2017. Language-level persistency. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. ACM, New York, NY, 481–493. DOI: https://doi.org/10.1145/3079856.3080229

[27] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-performance transactions for persistent memories. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*. ACM, New York, NY, 399–411. DOI: https://doi.org/10.1145/2872362.2872381

[28] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. 2016. Delegated persist ordering. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE, Los Alamitos, CA, Article 58, 13 pages. http://dl.acm.org/citation.cfm?id=3195638.3195709.

[29] Emre Kultursay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. 2013. Evaluating STT-RAM as an energy-efficient main memory alternative. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'13)*. 256–267. DOI: https://doi.org/10.1109/ISPASS.2013.6557176

[30] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, NY, 2–13. DOI: https://doi.org/10.1145/1555754.1555758

[31] J. Lee, T. Kim, and J. Huh. 2016. Reducing the memory bandwidth overheads of hardware security support for multi-core processors. *IEEE Transactions on Computers* 65, 11 (Nov. 2016), 3384–3397. DOI: https://doi.org/10.1109/TC.2016.2538218

[32] T. S. Lehman, A. D. Hilton, and B. C. Lee. 2016. PoisonIvy: Safe speculation for secure memory. In *Proceedings of the 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. 1–13. DOI: https://doi.org/10.1109/MICRO.2016.7783741

[33] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building durable transactions with decoupling for persistent memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. 329–343. DOI: https://doi.org/10.1145/3037697.3037714

[34] S. Liu, A. Kolli, J. Ren, and S. Khan. 2018. Crash consistency in encrypted non-volatile main memory systems. In *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. 310–323. DOI: https://doi.org/10.1109/HPCA.2018.00035

[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 *Proceedings of the 46th International Symposium on Computer Architecture (ISCA'19)*. ACM, New York, NY, 143–156. DOI: https://doi.org/10.1145/3307650.3322206

[36] Y. Lu, J. Shu, and L. Sun. 2015. Blurred persistence in transactional persistent memory. In *Proceedings of the 2015 31st Symposium on Mass Storage Systems and Technologies (MSST'15)*. 1–13. DOI: https://doi.org/10.1109/MSST.2015.7208274

[37] Y. Lu, J. Shu, L. Sun, and O. Mutlu. 2014. Loose-ordering consistency for persistent memory. In *Proceedings of the 2014 IEEE 32nd International Conference on Computer Design (ICCD'14)*. 216–223. DOI: https://doi.org/10.1109/ICCD.2014.6974684

[38] H. Mao, X. Zhang, G. Sun, and J. Shu. 2017. Protect non-volatile memory from wear-out attack based on timing difference of row buffer hit/miss. In *Proceedings of the Design, Automation, and Test in Europe Conference and Exhibition (DATE'17)*. 1623–1626. DOI: https://doi.org/10.23919/DATE.2017.7927251

[39] David Mulnix. 2015. Intel Xeon Processor D Product Family Technical Overview. Retrieved April 12, 2020 from https://software.intel.com/en-us/articles/intel-xeon-processor-d-product-family-technical-overview#_Toc419802876y.

[40] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. An analysis of persistent memory use with WHISPER. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. ACM, New York, NY, 135–148. DOI: https://doi.org/10.1145/3037697.3037730

[41] Moohyeon Nam, Hokeun Cha, Young Ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-optimized dynamic hashing for persistent memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. 31–44. https://www.usenix.org/conference/fast19/presentation/nam.

[42] Dushyanth Narayanan and Orion Hodson. 2012. Whole-system persistence. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. ACM, New York, NY, 401–410. DOI: https://doi.org/10.1145/2150976.2151018

[43] Yuanjiang Ni, Jishen Zhao, Daniel Bittman, and Ethan Miller. 2018. Reducing NVM writes with optimized shadow paging. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'18)*. https://www.usenix.org/conference/hotstorage18/presentation/ni.

[44] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16)*. ACM, New York, NY, 371–386. DOI: https://doi.org/10.1145/2882903.2915251

[45] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory persistency. In *Proceedings of the 41st Annual International Symposium on Computer Architecuture (ISCA'14)*. IEEE, Los Alamitos, CA, 265–276. http://dl.acm.org/citation.cfm?id=2665671.2665712.

[46] Joydeep Rakshit and Kartik Mohanram. 2017. ASSURE: Authentication scheme for SecURE energy efficient non-volatile memories. In *Proceedings of the 54th Annual Design Automation Conference (DAC'17)*. ACM, New York, NY, Article 11, 6 pages. DOI: https://doi.org/10.1145/3061639.3062205

[47] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. 2019. OneFile: A wait-free persistent transactional memory. In *Proceedings of the 49th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'19)*. 151–163. DOI: https://doi.org/10.1109/DSN.2019.00028

[48] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. 2015. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, 672–685. DOI: https://doi.org/10.1145/2830772.2830802

[49] B. Rogers, S. Chhabra, M. Prvulovic, and Y. 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 (MICRO-40)*. 183–196. DOI: https://doi.org/10.1109/MICRO.2007.16

[50] A. M. Rudoff. 2016. Deprecating the PCOMMIT Instruction. Retrieved April 12, 2020 from https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction.

[51] G. Saileshwar, P. Nair, P. Ramrakhyani, W. Elsasser, J. Joao, and M. Qureshi. 2018. Morphable counters: Enabling compact integrity trees for low-overhead secure memories. In *Proceedings of the 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-51)*. 416–427. DOI: https://doi.org/10.1109/MICRO.2018.00041

[52] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, and M. K. Qureshi. 2018. SYNERGY: Rethinking secure-memory design for error-correcting memories. In *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. 454–465. DOI: https://doi.org/10.1109/HPCA.2018.00046

[53] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin. 2017. Proteus: A flexible and fast software supported hardware logging approach for NVM. In *Proceedings of the 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50)*. 178–190.

[54] T. Silbergleit Lehman, A. D. Hilton, and B. C. Lee. 2018. MAPS: Understanding metadata access patterns in secure memory. In *Proceedings of the 2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'18)*. 33–43. DOI: https://doi.org/10.1109/ISPASS.2018.00012

[55] G. E. Suh, D. Clarke, B. Gasend, M. van Dijk, and S. Devadas. 2003. Efficient memory integrity verification and encryption for secure processors. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36)*. 339–350. DOI: https://doi.org/10.1109/MICRO.2003.1253207

[56] S. Swami and K. Mohanram. 2017. COVERT: Counter OVErflow ReducTion for efficient encryption of non-volatlle memories. In *Proceedings of the Design, Automation, and Test in Europe Conference and Exhibition (DATE'17)*. 906–909. DOI: https://doi.org/10.23919/DATE.2017.7927117

[57] Shivam Swami and Kartik Mohanram. 2018. ACME: Advanced counter mode encryption for secure non-volatile memories. In *Proceedings of the 55th Annual Design Automation Conference (DAC'18)*. ACM, New York, NY, Article 86, 6 pages. DOI: https://doi.org/10.1145/3195970.3195983

[58] S. Swami and K. Mohanram. 2018. ARSENAL: Architecture for secure non-volatile memories. *IEEE Computer Architecture Letters* 17, 2 (July 2018), 192–196. DOI: https://doi.org/10.1109/LCA.2018.2863281

[59] S. Swami, J. Rakshit, and K. Mohanram. 2016. SECRET: Smartly EnCRypted energy efficienT non-volatile memories. In *Proceedings of the 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC'16)*. 1–6. DOI: https://doi.org/10.1145/2897937.2898087

[60] Shivam Swami, Joydeep Rakshit, and Kartik Mohanram. 2018. STASH: Security architecture for smart hybrid memories. In *Proceedings of the 55th Annual Design Automation Conference (DAC'18)*. ACM, New York, NY, Article 85, 6 pages. DOI: https://doi.org/10.1145/3195970.3196123

[61] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. 2018. VAULT: Reducing paging overheads in SGX with efficient integrity verification structures. *ACM SIGPLAN Notices* 53 (March 2018), 665–678. DOI: https://doi.org/10.1145/3296957.3177155

[62] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. ACM, New York, NY, 91–104. DOI: https://doi.org/10.1145/1950365.1950379

[63] X. Wu and A. L. N. Reddy. 2011. SCMFS: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'11)*. 1–11.

[64] Jian Xu and Steven Swanson. 2016. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. 323–338. https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu.

[65] Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. 2006. Improving cost, performance, and security of memory encryption and authentication. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA'06)*. IEEE, Los Alamitos, CA, 179–190. DOI: https://doi.org/10.1109/ISCA.2006.22

[66] 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 *Proceedings of the 56th Annual Design Automation Conference (DAC'19)*. 1–6. DOI: https://doi.org/10.1145/3316781.3317869

[67] Mao Ye. 2018. Osiris: A low-cost mechanism to enable restoration of secure non-volatile memories. In *Proceedings of the 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-51)*.

[68] Vinson Young, Prashant J. Nair, and Moinuddin K. Qureshi. 2015. DEUCE: Write-efficient encryption for non-volatile memories. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. ACM, New York, NY, 33–44. DOI: https://doi.org/10.1145/2694344.2694387

[69] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. 2013. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. 421–432.

[70] 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 (ISCA'19)*. ACM, New York, NY, 157–168. DOI: https://doi.org/10.1145/3307650.3322252

[71] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-optimized and high-performance hashing index scheme for persistent memory. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 461–476. https://www.usenix.org/conference/osdi18/presentation/zuo.

[72] 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 (MICRO-52)*. ACM, New York, NY, 479–492. DOI: https://doi.org/10.1145/3352460.3358290

Received September 2019; revised December 2019; accepted February 2020