Triad-NVM: Persistency for Integrity-Protected and Encrypted Non-Volatile Memories

Amro Awad*
University of Central Florida
Orlando, Florida, USA
amro.awad@ucf.edu

Mao Ye
University of Central Florida
Orlando, Florida, USA
mye@knights.ucf.edu

Yan Solihin
University of Central Florida
Orlando, Florida, USA
yan.solihin@ucf.edu

Laurent Njilla
Air Force Research Laboratory
Rome, New York, USA
laurent.njilla@us.af.mil

Kazi Abu Zubair
University of Central Florida
Orlando, Florida, USA
kzubair@knights.ucf.edu

ABSTRACT

Non-Volatile Memory is here and provides an attractive fabric for main memory. Unlike DRAM, non-volatile main memory (NVMM) retains data after power loss. This allows memory to host data persistently across crashes and reboots, but opens up opportunities for attackers to snoop and/or tamper with data between boot episodes. While memory encryption and integrity verification have been well studied for DRAM systems, new challenges surface for NVMM if we want to simultaneously preserve security guarantees, data recovery across crashes/reboots, good persistence performance, and fast recovery.

In this paper, we explore persistency of data with all security metadata (counters, MACs, and Merkle Tree) to achieve secure persistency. We show that to ensure security guarantees, message authentication code (MAC) and Bonsai Merkle Tree (BMT) need to be maintained, in addition to counters, and they provide the majority of persistency overheads. We analyze the requirements for achieving secure persistency for both persistent and non-persistent memory regions. We found that the non-volatility nature of memory may trigger integrity verification failure at reboot, hence we propose a separate mechanism to support non-persistent memory region. Fourth, we propose designs to make recovery fast. Our evaluation shows that the proposed design, Triad-NVM, can improve the throughput by an average of 2× relative to strict persistence. Moreover, Triad-NVM can achieve orders of magnitude faster recovery time compared to systems without security metadata persistence.

CCS CONCEPTS

• **Security and privacy** → **Cryptography**; **Systems security**; *Database and storage security*; • **Computer systems organization** → **Architectures**; • **Hardware** → *Emerging technologies.*

*Most of this work was done as a part of Amro Awad's summer faculty fellowship at the U.S. Air Force (AFOSR SFFP) in summer 2018.

KEYWORDS

Security, Non-Volatile Memories, Persistence, Persistent Security

ACM Reference Format:
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 *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3307650.3322250

1 INTRODUCTION

Non-volatile main memory (NVMM) is here and provides a compelling set of features vs. DRAM. The recently released Intel Optane DC Persistent Memory provides a high capacity of up to 3TB/socket, low cost per bit, byte addressability, and non-volatility [1]. Since data remains in NVMM without power, data remanence is a real security vulnerability, hence Optane DC came with hardware memory-side encryption [2]. While memory-side encryption [3] may provide a good first line of defense as a standalone solution, its security protection is weaker due to not providing a secure execution environment to applications. For a more comprehensive protection, processor-side memory security is necessary.

NVMM also enables persistent applications which can recover after a crash or reboot by keeping data persisted in memory. While persistency and security may seem orthogonal, they should be studied together in NVMM as the system should be able to recover securely and guarantee its data integrity and confidentiality across crashes/reboots. However, persistency and security have largely been investigated separately; data recovery of memory across crashes is not a feature expected of volatile memory such as DRAM, except under cold boot attacks [4–6]. In such a system, security metadata is expected to be reinitialized at boot time [7]. Furthermore, recent studies in persistency and security focused on performance of normal operation without considering persistency across crashes and reboots [8–11]. Finally, memory persistency and crash recovery in NVMM are active research topics [12, 13], however they focus on persistency issues instead of security [14–17].

We argue that due to data remanence vulnerabilities, persistency and memory security require a comprehensive consideration of the following requirements:

- **Preservation of security guarantees**. Memory encryption and integrity verification should guarantee confidentiality and integrity of data, even with memory non-volatility.

- **Data recoverability**. NVMM may be divided into persistent and non-persistent regions, where the programmer keeps permanent data in the persistent region and temporary data in the non-persistent region. Data in a persistent region should be recoverable across crashes and system reboots. At the same time, NVMM should not get in the way of not wanting to recover data from non-persistent region.
- **Fast persistence and recovery**. Persistence should not significantly slow down execution. Post-crash/reboot recovery should be relatively fast to ensure high availability.

Current studies do not meet all said requirements. For example, recent studies explored atomically persisting data and encryption counters on NVMM [18–20]. While they provide data recoverability, they ignored the integrity protection of data and counters, which is essential for counter-mode memory encryption security [21]. Consequently, security guarantees are not preserved as the system is vulnerable to direct tampering of data, and counter replay attack [22], where an old version of both data and its counter are replayed to evade the encryption. Furthermore, the studies also did not discuss how persistent and non-persistent regions of memory must be treated differently, and did not discuss the issue of recovery time.

This paper presents the first work in comprehensively providing persistency in the context of a system with memory encryption and integrity verification. We discuss the problem of persisting security metadata in a system with NVMM employing memory encryption and integrity verification. Later, we define a new concept, ***secure persistency***, which defines the requirements for *securely* recovering integrity-protected and encrypted memory systems. Finally, we discuss several relaxation schemes and a comprehensive design that leverages them to optimize performance.

We make the following contributions in this paper. First, we show that persistent and non-persistent regions in NVMM must be handled differently in term of security metadata. Second, to ensure security guarantees, metadata such as message authentication code (MAC) and Bonsai Merkle Tree (BMT) need to be maintained, in addition to counters. Third, BMT needs to be updated from leaf to root every time new data value is persisted in NVMM, otherwise data can be recovered but its integrity can no longer be guaranteed post-recovery. In fact, persisting BMT metadata carries much higher overhead than persisting counters, so prior studies [18–20] under-estimated the overheads of persisting data in secure memory. Fourth, for non-persistent data, we found that the non-volatility nature of memory may cause integrity verification failure at reboot, hence we propose a separate mechanism to support non-persistent memory region. Fifth, we show that recovery may take more than one hour with TBs of NVMM hence we propose *lazy recovery* optimizations for making recovery fast.

To evaluate our design, we use Gem5 [23], a full-system cycle-accurate simulator. We use Linux kernel version 4.14 along with a disk image based on Ubuntu 16.04. Additionally, we initialize the kernel to dedicate the last 4GB (out of 16GB) as a persistent region, which we later mount as a directly-accessible (DAX) ext4 filesystem. The persistent region can be used by any library that supports persistent memory, however, we opt for using Intel's PMDK library [24] (previously known as PMEM) to build three microbenchmarks, in addition to four DAX-based synthetic workloads, along with 12 benchmarks from the SPEC2006 suite [25]. We additionally evaluate four workloads that run combinations of persistent (PMDK and DAX) and non-persistent (SPEC) workloads. Our simulation results show that Triad-NVM can improve the throughput by an average of 2× (relative to strict persistence). Moreover, Triad-NVM maintains a recovery time of less than 4 seconds for an 8TB NVM system (30.6 seconds for 64TB), which is 3648× faster than a system without security metadata persistence.

The rest of the paper is organized as follows. First, we discuss the problem and background of secure NVM systems in Section 2, where we also discuss the conventional way of persisting security metadata in secure NVMs. Later, in Section 3, we discuss the requirements of secure persistent memory systems. Later, we discuss different relaxation schemes and their impact on security, performance and resiliency of NVM systems. In Section 4, we discuss our evaluation methodology. Our results and discussion of our evaluation are presented in Section 5. Section 6 discusses the most relevant work to our paper. Finally, we conclude our work in Section 7.

2 BACKGROUND AND MOTIVATION

We will start our section with background discussion followed by discussing the issue of crash consistency of security metadata. Later, we present motivational data of the impact of the problem.

2.1 Background

Non-Volatile Main Memory (NVMM), such as Micron 3D XPoint or Intel DC Persistent Memory [26], are just around the corner and are expected to be on the market soon. They feature higher density, lower cost per bit, and lower idle power than DRAM, and non-volatility [27–29], but higher access latency and limited write endurance. Due to their non-volatility, they can be used as storage to host a filesystem, or as memory either persistent or non-persistent. Specifically, NVM-based DIMMs can be used to hold files and regular memory pages, and can be accessed in a way similar to DRAM through load/store operations. To realize this, new Operating Systems (OSes) started to support configuring the memory as persistent and conventional non-persistent memory. For example, recent Linux kernels and Windows started to support direct-access (DAX) support for filesystems [30]. In DAX-supported filesystems, e.g., ext4 with DAX, a file can be directly memory-mapped and accessed through typical load/store operations without the need to copy its pages to page cache as in conventional filesystems. For example, NVMM can be configured to have a partition dedicated to hold a filesystem. In Linux, when initializing the kernel, the parameter `memmap=4G!12G` can be used to dedicate the 4GB starting from address 12GB to directly-accessible filesystems. Thus, a single physical memory space may host persistent region (for files) and non-persistent region (for conventional memory pages).

2.1.1 *NVM Security.*

Due to non-volatility, NVMs are vulnerable to passive confidentiality breach (e.g. through data remanence attacks) as well as active integrity breach through data tampering. Non-volatility gives attackers plenty of time to perform both passive and active attacks both during operation, but most importantly during the off cycle between powered on episodes. Accordingly, NVMs

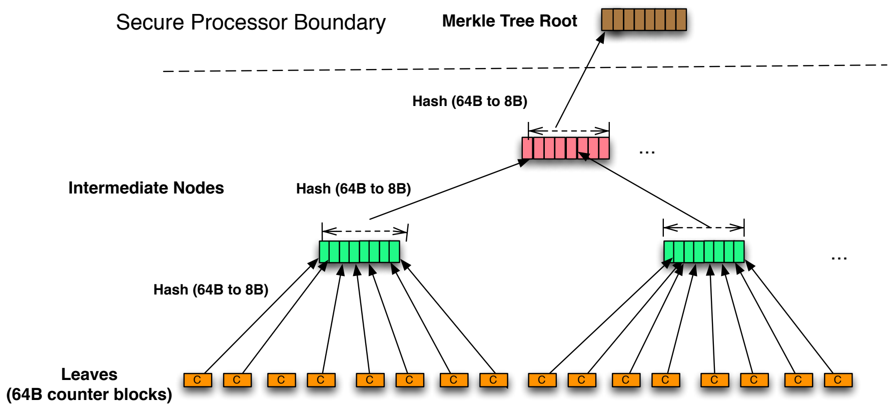

Figure 1: Example Merkle Tree.

must be coupled with memory encryption and integrity verification. Encryption may be performed at the memory side [3] or at the processor side [7]. Memory-side encryption is a standalone solution for memory products that relies on the memory as the root of trust. It embeds a crypto engine that encrypts data before it is written to memory row and decrypts it before sending it to the processor. However, its protection is weaker. Data is still communicated in plaintext on the system bus, exposing it to confidentiality breach. Establishing secure communication channel with the processor requires a complex authentication process [31]. It also does not provide secure execution environment for applications running on the processor as it cannot distinguish the legitimacy of access requests from the processor. Processor-side memory encryption assumes the processor chip as security boundary, thus eliminating said drawbacks, but it requires processor modifications. Data is encrypted prior to leaving the processor chip, and when it is fetched from NVMM, it is decrypted and its integrity is verified.

2.1.2 Memory Encryption and Integrity Verification. In counter-mode encryption, the encryption algorithm, e.g., AES, takes *initialization vector* (IV) as its input to generate a one-time pad (OTP) as shown in Figure 2. Later, when the block arrives at the processor chip, a low-cost bitwise XOR with the pad (encrypted IV) is performed to obtain the plaintext. By doing so, the decryption latency is overlapped with the memory access latency. In addition to its performance advantages, counter-mode memory encryption also provides defenses against a wide range of attacks. Counter-mode encryption is secure against dictionary-based attacks, known-plaintext attacks, and bus snooping attacks.

There are several possible counter formats, including monolithic (used in SGX) and split formats. Monolithic uses one (global) counter for all memory blocks, it is incremented each time any block is written back to memory to avoid counter reuse. An overflow of the global counter requires a costly entire memory re-encryption. Thus, monolithic counters need to be large, e.g. 64-bit. Since the counter value used to encrypt a block needs to be kept and retrieved for future decryption, large counters also require large on-chip cache to lower counter cache miss rates. In split-counter scheme, the encryption counters are organized as major counters (shared between cache blocks of the same page) and minor counters that are specific

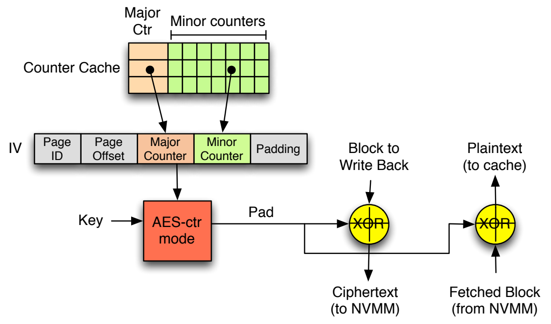

Figure 2: State-of-the-art split counter mode encryption from [9].

for each cache block [7]. Such organization allows packing 64 cache blocks' counters into a 64B block; 7-bit minor counters and 64-bit major counter. A major counter is incremented when one of its minor counters overflows, in which all corresponding minor counters will be reset and the whole page will be re-encrypted using the new major counter [7]. A major counter is large enough (e.g. 64-bit) that it does not overflow in the life of the system. Split-counter scheme is much more space efficient than monolithic counter scheme and achieves much higher counter hit rates, hence it performs significantly better than monolithic counter scheme. Thus, we assume a split counter scheme, just as assumed in recent studies [7–9, 31].

The fundamental security guarantee of counter-mode encryption requires that counter values cannot be repeated, or else the encryption can be defeated [21]. To avoid counter replay, counters must be protected against tampering through a Merkle Tree, referred to as Bonsai Merkle Tree or BMT [22]. This is an aspect ignored in prior studies [8, 18, 20]. With a BMT, data only needs protection through MAC but not a Merkle Tree. As shown in Figure 1, each encryption counter block, which is associated with a group of data blocks, is used to calculate a hash value (MAC) that will be used along with other hash values from other groups of counters to create a lower level hash value. Finally, the resulting MAC value, which is called *root*, is always kept in the processor. Each time a counter is brought from the insecure area, e.g., memory module, it will be verified by calculating the upper hash values and see if the result matches the root kept in the processor. Furthermore, when the processor writes a memory block and updates the corresponding counter, the BMT parent node must be updated to reflect the most-recent change. The propagation of the modification to the root depends on the requirement of data recovery.

2.1.3 Impact of Persistency on Security. NVMM may host persistent data, which needs to be recoverable after crashes/reboots. Since encrypted data needs to be decrypted with the correct (latest) counter value that was used to encrypt it, when data is persisted in memory, its counter must also be persisted. Otherwise, upon a crash, data may not be recoverable if decrypted using a stale counter value. Liu at al. [18] made this observation and proposed schemes to atomically persist data and its counter, either strictly (at every persist), or at epoch or transaction boundary. The atomicity is required not only to guarantee recoverability, but also security of the encryption, because a stale counter in memory may be used to induce a pad reuse [19].

2.2 Security Metadata Crash Consistency

It is critical to not just persist data and its counter. As stated earlier, counter-mode encryption fundamentally relies on the integrity of counters to be kept. Therefore, Merkle Tree must also be considered for crash recovery. After a crash occurs, the system must be able to recover and restore its encrypted memory data along with its accompanying security metadata. Figure 3 discusses the logical steps needed for crash consistency as described by Ye et al.[19].

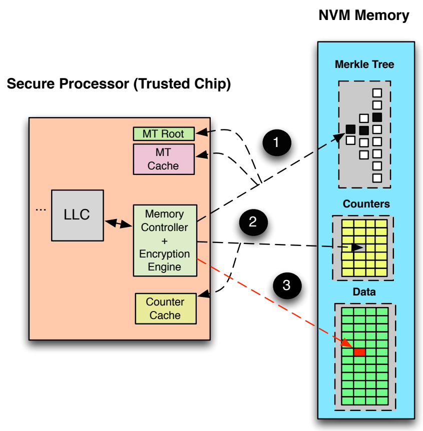

Figure 3: Description of write process that ensures crash consistency.

As depicted in Figure 3, the root of the BMT (on-chip) should be updated/persisted (Step (1)) along with affected intermediate nodes inside the processor, all the way to the root of the BMT. However, only the *root* of the tree needs to be kept in the secure region. In Step (2), the updated counter block is persisted (written to memory) as it gets updated in the counter cache. Finally, in Step (3), the written data block will be sent to the memory. Note that updating the root, counter and data can be done atomically through internal NVM registers or volatile registers that will be copied to internal NVM registers once a crash is detected[19].

2.3 Motivation

As we now understand the issue of guaranteeing security metadata crash consistency, let us now discuss the impact of ensuring such consistency on performance of NVMM. As discussed earlier, a secure persistent system would persist any changes to the BMT, encryption counter, data, and MAC on each memory write operation. When these operations are performed atomically, substantial slowdowns will be incurred.

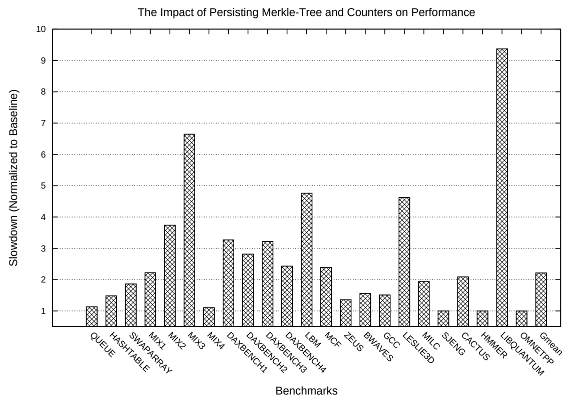

Figure 4: Performance Overhead of Persisting Security Metadata.

Figure 4 depicts the relative system throughput when using strict persistent security[1]. As shown in the figure, most workloads get severe performance degradation due to the impact of additional write operations to ensure persistency of security metadata. Notably, some workloads can be slowed down by as much as 9.4× compared to a system that does not guarantee such type of persistency. The impact of persisting security metadata depends mainly on the memory behavior and write-intensity of the running applications. Unfortunately, with such performance overheads, many users may avoid enabling persistent security which would leave their systems insecure and potentially their data unrecoverable. To address this issue, in this paper, we define the requirements of secure persistent systems and study the impact of different schemes on performance, recovery time and resilience of the system. To the best of our knowledge, our paper is the first to consider all security metadata on secure persistent memory.

3 DESIGN

In this section, we first analyze the requirements of persistently secure systems. Later, we discuss the different design options that can be adopted for the purpose of implementing persistently secure systems. Finally, we discuss our proposed design that combines several novel optimizations.

3.1 Threat Model

We assume a single processor chip system where the chip boundary forms the security boundary. The root of trust must be self contained in the processor chip, including any keys and root of the Merkle Tree.

[1]More details about the workload and methodology can be found in Section 4.

We assume attackers capable of performing passive reading of memory values from the NVMM and snooping the bus connecting processor and memory. We also assume attackers capable of changing values stored in the NVMM, either while the computer system is powered on and operational, or across episodes of power events. Attackers may achieve that either through physically tampering the memory (e.g. using heat gun or electric currents) or by unmounting it from the target system and mounting it to a different system. However, our goal is to ensure confidentiality and integrity of memory data, but not its availability. Hence, denial of service is not a focus of this paper.

We also assume that the entire main memory is protected, not just a small part of it that is within a security enclave. Hence, our protection goal is different from that of Intel SGX, and requires the entire memory to be protected.

3.2 Requirements for Secure Persistence

Our first contribution is an analysis of the requirements of secure NVMM. NVMM will be divided into a persistent region which hosts persistent data (memory data structures or a part of the file system) and non-persistent data. The first requirement for a secure persistent memory is the **preservation of security guarantees** of confidentiality and integrity of data. This applies to both persistent and non-persistent memory regions. For a counter-mode memory encryption, this requirement means that data is encrypted using counter-mode, data is protected using MAC, and counters are protected using a Bonsai Merkle Tree (BMT), where the root is kept persistently on chip. Furthermore, the operation of data and security metadata should be such that ① The same counter value for a block cannot be reused without changing the key, ② the BMT root must reflect the most-recent state of the memory.

A second requirement is **data recoverability**, which applies differently to persistent and non-persistent regions. Data in a persistent region should be recoverable across crashes and reboots, and security metadata (counters, MACs, and BMT nodes) should be persisted in such a way that enables recovery. We argue that to support data recovery across crashes or reboots, any persisted data must also have its latest security metadata (counter, MAC, and BMT) persisted. However, different types of metadata need persistence for different reasons. The persisted latest counter enables data to be decrypted to the correct value. The persisted latest MAC value allows data tampering to be detected. Specifically, without persisting the latest MAC value, an attacker may roll back data to a stale version that matches the stale MAC, without being detected. BMT must also reflect the latest counter value in order to detect any attempts to replay old counter values. However, we note that only the root needs to be persisted, because intermediate BMT nodes can be re-calculated. Therefore, persisting data requires a significant cost of ensuring atomic update of data and security metadata. We refer to this requirement as *strict persistency of security metadata.*

Atomicity of multiple updates (to data, counter, MAC, and BMT) can be achieved through transactional mechanism such as hardware logging [32], or a lighter weight mechanism utilizing the write pending queue (WPQ) in the memory controller (MC) [18, 33]. We rely on the latter mechanism to atomically persist data and security metadata in this work.

The hierarchical implementation of a BMT reduces the verification overhead of a read operation. If a counter is read from memory, if its parent node has been continuously cached in the processor chip, i.e., has been already verified, then there is no need to verify further. Similarly, if its grandparent node is present but its immediate parent is not, then only the immediate parent needs to be fetched from memory and verified against the grandparent while the counter/leaf is verified against the fetched parent. In the worst case, when none of a counter's ancestor nodes are found on chip, all nodes must be fetched and integrity verification must be performed from leaf to root. However, given the spatial locality of most applications, they rarely need to access more than few levels of ancestors.

The write operation also has a low overhead if the main memory is volatile. When a counter value is updated, its parent node is updated (if on chip), but all subsequent ancestor nodes do not need to be. If the parent node is evicted from the cache at a later time, the grandparent node (if on chip) is updated before the parent node is evicted. Thus, BMT modifications are slowly propagated from leaf to root lazily based on eviction of BMT nodes. However, for NVMM this lazy update violates recoverability because when power is lost, values of the intermediate BMT nodes are lost before being reflected at the root, resulting in a stale root. A stale BMT root prevents counter integrity to be verified and may permit a counter replay attack to go undetected. Therefore, in NVMM, a write operation must be handled differently. When a counter is updated, under strict persistency all its ancestors must be updated all the way to the root of the BMT. This is expensive. As NVMs are expected to have huge capacities, e.g., tens of TBs, the height of the Merkle Tree is expected to be in order of tens of levels, which would require updating tens of ancestor node blocks. Later, we will present an optimization to accelerate the non-persistent region portion of the BMT.

Data recoverability for non-persistent region of the NVMM has a different requirement. Non-persistent data does not need to be recovered after crashes/reboots. Hence, it is not important to apply strict persistency to writes to NVMM. We can let NVMM writes to data, counters, MACs, and BMT nodes to occur asynchronously (without atomicity). After a crash/reboot, we can assume that data is discarded and all security metadata will be rebuilt. However, this introduces a new dilemma. Due to mismatch of data and security metadata in the NVMM, the first access by the processor to data after a crash/reboot is likely going to trigger an integrity verification failure. To avoid that, we can incorporate into the boot process an initialization of non-persistent memory region and during this time rebuild all security metadata. However, NVMM is expected to have a huge capacity (TBs), the boot process may take an unacceptable latency.

This brings us to the last requirement of **fast persistence and recovery**. Persistence should not significantly slow down execution. At the same time, post-crash/reboot recovery should be relatively fast, even for non-persistent memory region. In the following sections, we will discuss our performance optimizations to the just-discussed basic secure persistent memory design that we refer to collectively as Triad-NVMM.

3.3 Triad-NVMM Design

3.3.1 Persistent and Non-Persistent Regions. In current systems, at boot time, the kernel is initialized with a value that determines which range of the memory address is considered persistent. Such region can be formatted and mounted as a filesystem, e.g., ext4 filesystem. Additionally, such persistent region can be used by persistency APIs, such as PMDK, which would backup the persistent data structures by files in the persistent region. Any file in the persistent region can be memory-mapped, and accessed through load/store instructions. Meanwhile, the non-persistent region in the NVM device is not ensured recoverability after a crash; an application will only be able to access data persisted to the persistent region after recovery. Figure 5 depicts a system with encryption counters of persistent and non-persistent regions in an NVM device. Note that such distinction between persistent and non-persistent memory regions are critical to how security metadata should be handled, since the regions have different data recovery requirements.

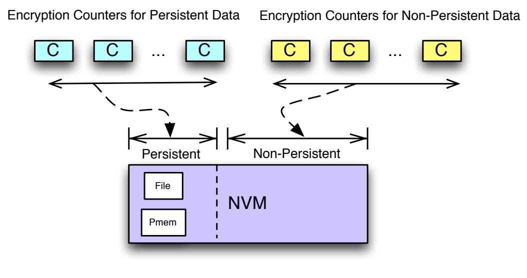
Encryption Counters for Persistent Data
Encryption Counters for Non-Persistent Data
C
C
...
C
C
C
...
C
Persistent
Non-Persistent
File
Pmem
NVM

Figure 5: Encryption counters for data with different persistency requirements.

We note that prior work [18] exposes persistency model to hardware in order to relax the persistency of data and its counter. For example, data and counter are persisted only at epoch or transaction boundaries. The cost to such an approach is that an architecture mechanism must be there to bridge a high-level programming language construct (memory persistency) with low level security metadata mechanism that used to be transparent to software. While we choose a different approach, we note that it is orthogonal to our work, and can be added to our mechanism if desired.

Just as the counters being separate for persistent and non-persistent regions, MACs protecting data are also separate. The security metadata (both counters and MACs) for the persistent region is allocated in the region itself, while that for non-persistent region is allocated in the non-persistent region. This way they can be handled and processed separately solely by their addresses.

While BMT covers all encryption counters, regardless of their persistence requirements, not all parts of the BMT need to be persisted. Specifically, given that current systems define the persistent regions of the NVM at boot time, there is a range in memory that is guaranteed not to be used for persistent data. Based on such an observation, a BMT can be divided in two different ways. One way to divide the BMT is to still use a single BMT but split it into multiple subtrees, i.e., some parts only cover the first 4GB of the main memory as aforementioned, hence only the persistent-region subtree needs to be persisted strictly. This is illustrated in Figure 6.

As depicted in Figure 6, when an update occurs for a persistent memory location, all the corresponding parts for such location in BMT at all levels should be updated up to the root (inside the processor chip). Updates to BMT for persistent data should update the following persistently (in memory or internal NVM registers) and in cache (if cacheable) ① the root of the Merkle Tree, ② the root's child node that owns the updated counter, and all other intermediate nodes corresponding to the updated counter (as in ② and ③). Finally, as in ④, the counter is persisted in both the memory and counter cache. However, for non-persistent data, updating the BMT intermediate nodes and counters in the cache and lazily write them back to memory as they are naturally evicted from the cache, is sufficient. Clearly, the subtrees that belong to persistent regions should be separable, i.e., some parts of the root only belongs to persistent region and others only to non-persistent regions. Having some parts belonging to both is challenging and can lead to unverifiable parts at recovery; that part will reflect most-recent values of both types of data, but we will be unable of reproducing the root due to the lost updates of intermediate nodes and counters of non-persistent data. The only level that has MAC values on the same block for both persistent and non-persistent data is the root, however, they are clearly separable. To avoid cases of a BMT node covering both types of data, the memory space ratio between persistent and persistent-data should 0:8, 1:7, 2:6, 3:5, 4:4, 5:3, 6:2, 7:1 or 8:0. In other words, each MAC value in the root should cover either persistent or non-persistent data but not both. Note that the root is kept in a persistent register in the processor, hence it does not need to be updated in memory.

Another way to split the BMT for persistent and non-persistent regions is to have two completely separate BMTs with separate roots. With separate BMTs, there is no concern for having to keep fixed ratio of capacities between persistent and non-persistent regions. Furthermore, this allows persistent region BMT to be stored in the persistent region itself. Likewise, the non-persistent region BMT can also be stored in the non-persistent region. This makes it consistent with the separation of counters and MACs for the two region and allows the security engine to treat security metadata differently based on their physical addresses. Thus, we chose this approach.

3.3.2 Relaxing Encryption Counter Persistency. As we now understand how future NVM devices will be leveraged by the OS, let us now discuss how encryption counters should be treated differently based on persistency of the regions. While encryption counters of persistent data must be recovered correctly for both correctness and security reasons, those of non-persistent data can be relaxed from ensuring correctness after recovery. As mentioned earlier, for non-persistent data, we can relax strict persistency for data and security metadata, which will boost performance for accessing such data.

Note that, however, despite not needing data recovery, non-persistent region security has a unique vulnerability different from volatile memory. In particular, counter replay can be performed across boot episodes, not just within a boot episode. Relaxing strict persistency of security metadata for non-persistent data introduces a security vulnerability of across-boot counter reuse. To illustrate this, consider data that has persisted but its corresponding counter

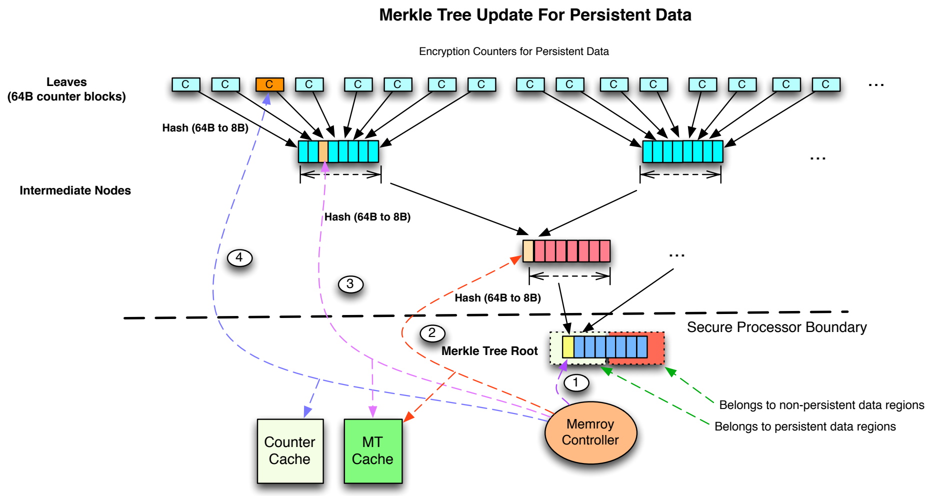
Merkle Tree Update For Persistent Data
Encryption Counters for Persistent Data
Leaves (64B counter blocks)
C
Hash (64B to 8B)
Intermediate Nodes
Hash (64B to 8B)
Hash (64B to 8B)
4
3
2
1
Secure Processor Boundary
Merkle Tree Root
Belongs to non-persistent data regions
Belongs to persistent data regions
Counter Cache
MT Cache
Memroy Controller

Figure 6: Updating Merkle for counters correspond to data with different persistency requirements.

value has not. If power is lost at this point, the most recent counter value is lost, and we are left with a stale counter value in the NVMM. On a reboot, the stale counter value is read from memory and thus counter reuse has occurred. This violates the security requirement of counter-mode encryption. To avoid counter reuse across boot episodes, we investigate two schemes. The first scheme relies on the use of two different keys: *Persistent Key* and *Volatile Key*. The persistent key is used for encrypting/decrypting persistent data regions, whereas the volatile key is used for non-persistent data. The volatile key is changed at reboot, which avoids counter reuse even as we relax strict persistence of encryption counters of non-persistent data. Our design only requires modifying the memory controller to be able to receive a command from the kernel which notifies the MC about the start and end address of the persistent region, and to use such information to decide which key to use. Moreover, the kernel needs to send the memory controller, through a memory-mapped register, a command to change the volatile key at the time of recovery or reboot.

An alternative scheme is to use a single key but add a session counter to the encryption IV, in addition to other components of the IV (major and minor counter, page ID, page offset, etc.). Persistent data uses session 0 because it needs to maintain data recovery across reboots, but non-persistent data uses a non-zero session counter that is incremented at each boot. The session counter approach reduces the complexity of having to generate and manage multiple security keys, as only one key can be used. More importantly, however, it will allow us to precompute the BMT root for the next session to enable fast recovery of the BMT (this will be discussed later). The current session counter value must be kept persistently on chip along with BMT roots. Due to its advantages, we chose this approach.

3.3.3 *Relaxing Merkle Tree Persistency.*

As we discussed in the previous part on how to divide BMT vertically into persistent and non-persistent subtrees, we now will discuss which levels of Merkle Tree need to be updated and the impact of possible relaxation schemes.

Unlike encryption counters, BMT intermediate nodes can be rebuilt if lost and their main use is to speed up verification and updating the root of the BMT. However, if encryption counters are guaranteed to be strictly persistent and up-to-date, then after a crash it is possible to rebuild all levels of intermediate nodes all the way up to the root which needs to match the root value in the processor. While this looks like a straight-forward optimization, it can actually create a single-point of failure; if any counter has been corrupted, e.g., due to memory error, we can only know that the Merkle Tree root has not matched and thus none of the memory is integrity-verifiable. While a Merkle Tree root can hold 64B instead of only 8B values, an uncorrectable counter error would still result in $\frac{1}{8}$th of the memory being lost/unverifiable. As NVMs are expected to contain terabytes of data, the chance that an uncorrectable error of any counter can lead to large part of memory being unverifiable is unacceptable. Moreover, only guaranteeing the persistence of encryptions would require high recovery time due to the need of iterating over all encryption counters to rebuild all levels of Merkle Tree.

To enable high-resolution of identifying unverifiable locations in addition to keep the recovery time manageable, we propose persisting the first N levels of the Merkle Tree (from leaf up) while optionally maintaining several NVM registers within the processor. The value of N depends on the acceptable performance/recovery-time trade-off. Persisting low levels of Merkle Tree can help isolating problems and identifying which counters are corrupted/uncorrectable. Moreover, more number of levels being guaranteed persistence allows shorter recovery time to reconstruct the Merkle Tree. Also note that since higher levels of the Merkle Tree has much less intermediate nodes, persisting them reduces the chances of the inability to construct Merkle Tree due to uncorrectable errors.

3.3.4 *Non-Persistent Region Crash Recovery.*

For post-reboot recovery, the BMT has to be reconstructed again to be able verify any tampering. During recovery, before admitting the current status of the system as verified, all persistent locations need to be verified through reconstructing the corresponding parts of the Merkle Tree and verifying that it generates a correct root. This ensures that the data has not been tampered with from (or before) crash time till recovery time. However, for non-persistent data locations, while we do not need to verify the integrity of data at the recovery time, we need to be able to verify it after recovery. Recall the dilemma of a mismatch of data and security metadata in the NVMM on the first access by the processor to data after a crash/reboot triggering an integrity verification failure, even when we do not wish to recover data or security metadata. To avoid that, we can incorporate into the boot process an initialization of non-persistent memory region by zeroing all of them out, and during this time rebuild all security metadata, including incrementing a session counter, zeroing of all other counters, recalculating all MACs, and rebuilding the BMT. However, NVMM is expected to have a huge capacity (TBs), the boot process may take an unacceptable latency. To better understand the overhead, let us assume a 6TB of NVM, where 50% is used as non-persistent data, i.e., normal memory. At recovery time, if reading each data block and initializing it takes 100*ns*, then just iterating over the 3TB (non-persistent region) would take 5154 seconds (almost 85.9 minutes). This is in a stark contrast to the persistent data region (the other 3TB), where low level nodes of the BMT along with data and counters are strictly persisted. Hence, only the upper levels of Merkle Tree need to be reconstructed. For instance, if only the first level of Merkle Tree (the parents of counter blocks), and reading each block and calculating its MAC values takes 100*ns*, then that will take only 92 seconds (1.5 minutes). Thus, we wish to make the reboot process for non-persistent region significantly faster.

To avoid an unanticipated long recovery time of rebuilding counters and Merkle Tree parts of non-persistent region, we propose a *Lazy Recovery* method. As the number of counter and data blocks is the largest in the lower level and they consume most of the rebuilding time, we can lazily update them as following. By initializing all intermediate nodes at level 1 (parents of counter blocks) with zeros, and constructing all upper levels sequentially, we can obtain an initial root value (or part of root) for non-persistent data. Later, any update to any counter value, if the parent intermediate node has a zero value, we do not flag an error or tampering as the upper levels of Merkle Tree (and root) are already updated to reflect such value, however, we know that this is the first write to the counter block after recovery and hence we zero out (or initialize) the counter value and update the parent node accordingly. Note that since each counter block has its MAC value stored in 8 bytes of the 64B parent node, only the corresponding 8B in the parent node would indicate if it is the first update after recovery or not. While the odds that a counter block value would naturally lead to 64-bit zero MAC value is only $\frac{1}{2^{64}}$, to avoid falsely assuming an initialized counter value, if a counter block naturally has a MAC value of 0, we increment one of its minor counters, re-encrypt the cacheline corresponds to that counter, and calculate its new MAC value which will be used in the parent node if does not equal zero. By lazy update of non-persistent data and counter blocks, we ensure a recovery process that only needs to iterate over levels 1 or 2 instead of all corresponding data and counter blocks.

3.3.5 Overall Design. Figure 7 depicts the Triad-NVM design. As mentioned earlier, Write-Pending Queue (WPQ) is considered part of the persistence domain (power-fail protected domain) in modern processors [34]. Thus, anything that reaches there should be consistent or there is a way to ensure its consistency. To do so, each write operation, before being persisted (removed from volatile buffer), it logs all its corresponding updates (counter, data, MT nodes and root) to persistent registers inside the processor and then set a persistent bit (called *READY_BIT*). If a crash occurs while copying the updates in persistent registers to WPQ, then when the system restores the memory controller will attempt to write the persistent registers to NVM (or WPQ) again. At the time of copying the contents from the registers to WPQ, Triad-NVM selectively chooses if the counter or Merkle Tree nodes should be copied to WPQ or it is just enough to update them in caches (as shown in steps ⑧ and ⑨). Note that once a dirty block gets evicted from these caches it will go to WPQ as usual. Persistent registers can be implemented as fast NVM registers or volatile registers will be flushed to slower NVM registers once a crash occurs through leveraging ADR or residual power. The number of persistent registers depends on the Triad-NVM model, e.g., if TriadNVM-2 is used then we only need 5 registers, whereas if we use the impractical strict persistence then we might need up to 15 registers. Note that using persistent registers have been also assumed in state-of-the-art work [19].

If the updated encryption counter corresponds to a persistent memory region, it will be strictly updated (copied to WPQ). However, for Merkle Tree, if the updated parts belong to a persistent memory region and in a level higher than the *persist level*, then the updates will be persisted to NVM (copied to WPQ). Note that the persist level is the highest level of Merkle Tree that is guaranteed to be strictly persisted, e.g., if such a level is 2, then counters, their parents and grandparents are guaranteed to be persisted after each update. Note that TriadNVM recovery process is as simple as iterative over the intermediate notes at the persist level and construct the upper levels of the persistent memory regions, however, additionally initialize the intermediate nodes of such level to zeros for non-persistent memory regions before constructing their upper levels of the tree.

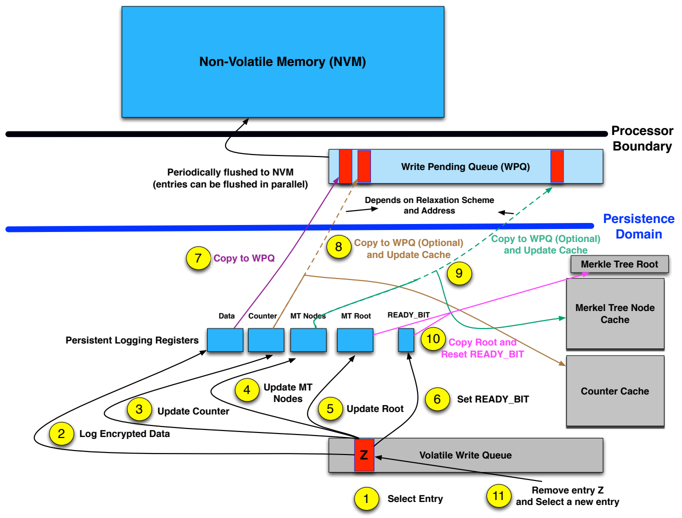
Non-Volatile Memory (NVM)
Processor Boundary
Write Pending Queue (WPQ)
Periodically flushed to NVM (entries can be flushed in parallel)
Depends on Relaxation Scheme and Address
Persistence Domain
7 Copy to WPQ
8 Copy to WPQ (Optional) and Update Cache
Copy to WPQ (Optional) and Update Cache
9
Merkle Tree Root
Merkel Tree Node Cache
Data
Counter
MT Nodes
MT Root
READY_BIT
Persistent Logging Registers
10 Copy Root and Reset READY_BIT
Counter Cache
4 Update MT Nodes
3 Update Counter
5 Update Root
6 Set READY_BIT
2 Log Encrypted Data
Z
Volatile Write Queue
1 Select Entry
11 Remove entry Z and Select a new entry

Figure 7: High-Level Overview of Write Operation on Triad-NVM.

4 METHODOLOGY

In this section, we describe our evaluation methodology. Since our work involves kernel modifications and both persistent and non-persistent applications, we use Gem5 simulator[23] with full-system mode. The kernel we simulate is based on Linux Kernel 4.14. Moreover, the disk image we use is based on Ubuntu 16.04 distribution. The kernel was initialized with configuring the 4GB starting from 12GB as a persistent region, e.g., the kernel was initialized with `memmap=4G!12G`. Later, the persistent regions was formatted with DAX-enabled ext4 filesystem and mounted for use by persistent applications and libraries. Table 1 presents the architectural configurations of our simulated system.

Table 1: Configuration of the simulated system.

Processor	
CPU	8-core, 1GHz, out-of-order x86-64
L1 Cache	private, 2 cycles, 32KB, 2-way, 64B block
L2 Cache	private, 20 cycles, 512KB, 8-way, 64B block
L3 Cache	shared, 32 cycles, 8MB, 64-way, 64B block
DDR-based PCM Main Memory	
Capacity	16 GB
PCM Latencies	60ns read, 150ns write [28]
Organization	2 ranks/channel, 8 banks/rank, 1KB row buffer, Open Adaptive page policy, RoRaBaChCo address mapping
DDR Timing	tRCD 55ns, tXAW 50ns, tBURST 5ns, tWR 150ns, tRFC 5ns [18, 28] tCL 12.5ns, 64-bit bus width, 1200 MHz Clock
Encryption Parameters	
Counter Cache	128KB, 8-way, 64B block
Merkle-Tree Cache	128KB, 8-way, 64B block
Merkle-Tree	9 levels, 8-ary, 64B blocks on each level

To evaluate the impact of our proposed optimizations, we run 3 sets of workloads: persistent applications, non-persistent applications and a combination of both. For the benchmarks being used, following is the description of the major applications followed by Table 2 which shows the mixed workloads we use. For non-persistent workloads, we use representative benchmarks from SPEC2006 [25]. We also implement persistent Hashtable, ArraySwap and Queue benchmarks using Intel PMDK library. Additionally, we implement synthetic benchmark, DAXBENCH-*S*-*RW* that leverages DAX to `mmap` a file directly in the persistent region and access it through memory load/store operations with *S* stride and *RW* read to write ratio. Finally, we use those benchmarks to create multi-programmed workloads to enable studying the optimizations that relax persistence of security metadata for non-persistent data applications.

Table 2: Mixed Benchmarks

Benchmark Combination	
DAXBENCH1	DAXBENCH-128-2
DAXBENCH2	DAXBENCH-1024-2
DAXBENCH3	DAXBENCH-256-2
DAXBENCH4	DAXBENCH-512-3
MIX1	Array-swap, Queue, Hashtable, DAXBENCH-64-2
MIX2	Mcf, Queue, Hashtable, DAXBENCH-64-2
MIX3	Mcf, LBM, Hashtable, DAXBENCH-512-2
MIX4	Array-swap, Hashtable, Hashtable, DAXBENCH-1024-2

All applications were fast-forwarded to representative regions and each run simulated at least 200M instructions.

5 EVALUATION

In this section, we evaluate our proposed optimizations and study the impact of combining them on Triad-NVM. We will first study the impact of each optimization on performance, and later study and analyze the trade-off between recovery-time and performance overhead when relaxing the persistence of Merkle Tree.

5.1 The Impact of Relaxation Schemes on Performance

Figure 8 shows the impact of Triad-NVM on performance. TriadNVM-*N* represents TriadNVM with strict persistence of persistent regions up to the *N*th level of the Merkle Tree. As expected, for the persistent region, TriadNVM-1 is expected to perform better than TriadNVM-2 and TriadNVM-3. The main reason for that is that less number of writes need to occur to memory to ensure strict persistence of Merkle Tree, i.e., in TriadNVM-1 only the parent of the updated counter block needs to be persisted along with the counter block, whereas in TriadNVM-2 both the parent and the grandparent of the updated counter block need to be persisted along with the counter block. Clearly, there will be no difference between different TriadNVM persistence levels for non-persistent workloads, e.g., LBM and MCF. Moreover, in mixed workloads, only the writes to persistent regions will make difference between TriadNVM models.

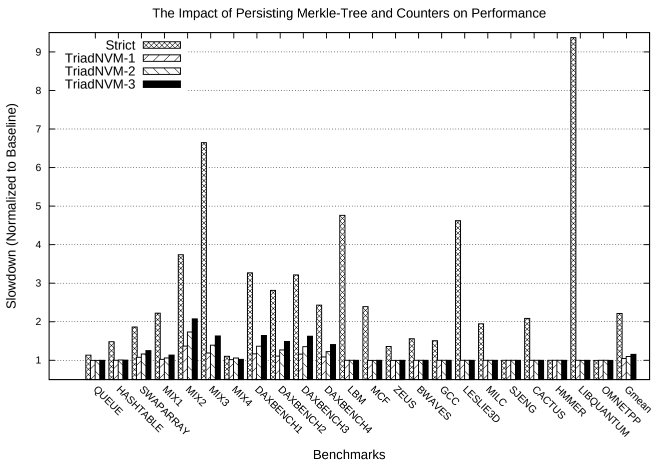

Figure 8: The impact of Merkle Tree persistence model on Throughput.

Our results show that using strict persistence scheme can lead to an average of 2.2× slowdown, whereas TriadNVM-1, TriadNVM-2 and TriadNVM-3 lead to only 4.9%, 10.1% and 15.6% performance overheads, respectively. Moreover, we observe that write-intensive workloads with non-persistent region allocations, e.g, libquantum, can get an almost order of magnitude speed up when using schemes that are aware of persistent regions and thus relax the requirements of non-persistent memory ranges.

As discussed earlier, NVMM suffers from slow writes and limited write endurance. Ensuring higher level of persistence in the BMT reduces the recovery time and avoids single-point of failures, but it may substantially increase the number of writes. On the other hand, more relaxed schemes, e.g., TriadNVM-1, increases recovery time but incurs a smaller number of extra writes. Therefore, choosing the right design point is important in balancing off recovery time and the number of writes to NVMM.

Figure 9, shows the total number of times in our simulations for different schemes for each workload. We can observe that most

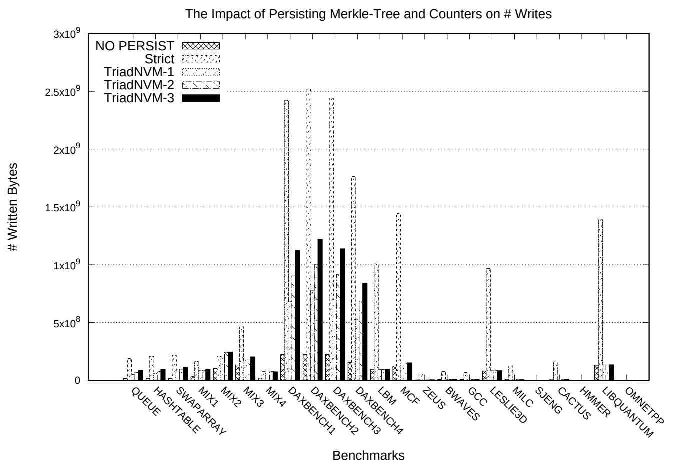

Figure 9: The impact of Merkle Tree persistence model on # Writes.

applications' # writes are proportional to the persistence level, however, since large percentage of writes could be resulting from natural cache write backs and data persistence, only the writes result from persisting security metadata would increase linearly with the level of persistence. We can observe that for most workloads, the number of writes for TriadNVM is close to the original number of writes without persisting security metadata.

5.2 The Impact on Recovery Time

As mentioned earlier, strict persistence ensures near zero recovery time, however, at the cost of significant performance degradation during normal operation. Meanwhile, not persisting security metadata at all will require creating Merkle Tree and initializing MAC values in data blocks which requires iterating over all memory blocks. In contrast, Triad-NVM guarantees the persistence of the counters and the first level of the Merkle Tree. To estimate recovery time, we assume that reading a tree block in addition to calculating its MAC value takes 100*ns*. Figure 10 shows the impact of persisting security metadata on recovery time.

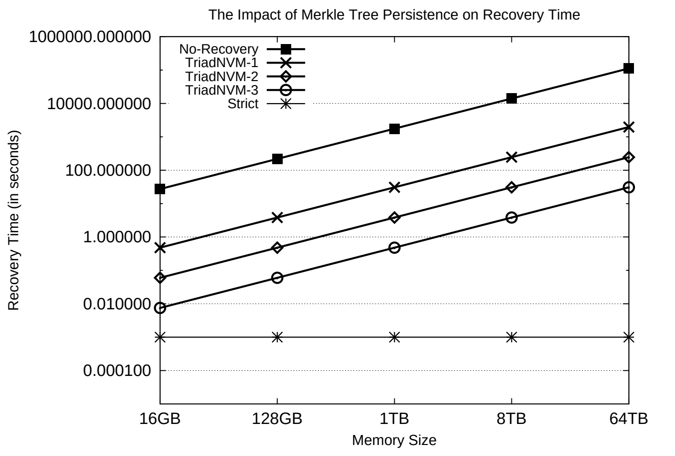

Figure 10: The impact of Merkle Tree persistence model on Recovery Time.

As emerging NVMs are expected to be multiple TBs in size, our goal is to get an estimation of how long does it take to recover the whole Merkle Tree for different persistency models. We expect TriadNVM-3 to have the least recovery time as it requires the least number of computations; starting the reconstruction from Level 3. Meanwhile, TriadNVM-2 brings in a tradeoff between better performance but almost 8× slowdown. In the case of strict persistency, there is almost zero recovery time; all parts of Merkle Tree are up to date. As shown in the figure, the problem becomes more obvious when we deal with terabytes of memory, e.g., at 1TB the recovery (construction) time for no-persist scheme is 30 minutes. In contrast, for TriadNVM-1 it takes only 30.68 seconds to complete recovery. As system downtime can be in hundreds of thousands per minute [35], some high-availability systems might trade performance for faster recovery time. For instance, if TriadNVM-2 or TriadNVM-3 are used with 1TB of memory, then the recovery times are 3.83 and 0.48 seconds, respectively. We can also observe that when the memory 8TB, the recovery time is almost 8x slower than that of 1TB, due to linear increase in number of nodes/blocks to iterate through to calculate MAC values. While TriadNVM can be configured to persist up to any specific value, we leave the decision of which level to persist to up to system integrator or administrator and based on the acceptable performance/recovery-time trade-off. Note that such a decision can be also affected by the crash rate and possibility of system failures in addition to how sensitive the data is and tolerance to declaring them unverifiable at recovery time.

As resilience of the system is also important, TriadNVM-2 and TriadNVM-3 provide a higher resolution of pinpointing errors. In TriadNVM-2, if calculating the MAC values starting from level 2 does not eventually generate the root value stored in the processor, then TriadNVM restarts form level 1 and find out which node in Level 2 does not match with that calculated from its children, and hence only declaring the corresponding 32KB as unverifiable. Note that the root must match with that generated from those in Level 1, before declaring that Level-2 node to be the source of error. Similarly, if an uncorrectable error occurs in Level-1 nodes, while Level-2 nodes ended up generating the same root value, i.e, verified, then we can use Level-2 nodes to pinpoint which Level-1 node is the corrupted one. TriadNVM-3 follows the same logic but now add an additional level of isolation in case uncorrectable errors occurred on both Level 1 and Level 2. It is also important to note that since the number of levels of Level-2 and Level-3 are 8 and 64 times, respectively, smaller than that of Level-1, then the chances of uncorrectable errors on them is also smaller.

6 RELATED WORK

Persisting security metadata has been rarely studied up until recently. Recent work by Liu et al. [18] pointed to the need of atomically persisting data and counters to achieve crash consistency. They proposed strict persistency, and also an optimization where data and counter are persisted atomically only at epoch or transaction boundaries. The atomicity mechanism utilizes the ADR platform feature that requires the write pending queue (WPQ) of the memory controller to be in the non-volatile domain. Zuo et al. [20] added a write reduction optimization by combining/coalescing multiple updates to different counters on a single counter block into a single write to memory. Another recent study by Ye et al. [19] proposed repurposing error-correcting code (ECC) for making a sanity-check for encryption counters. By doing so, strict persistence of an encryption counter can be relaxed; a counter value can be restored by trying several consecutive values until ECC match occurs. We assume strict counter persistence in our paper but note that the techniques discussed above [18–20] are orthogonal and can be applied to reduce NVMM writes in our case as well. A concurrent work to Triad-NVM is Anubis [36]. Anubis addresses the recovery time problem in secure NVMs by persistently tracking addresses in metadata caches. Triad-NVM tackles resilience, performance, recovering systems with both persistent and non-persistent regions, whereas Anubis is focused on reducing recovery time. Anubis achieves better recovery time but could introduce single-point of failure when some counters are corrupted. On the other side, Anubis address the problem of recovering parallelizable Merkle Trees, such as those in Intel's SGX.

One significant difference between prior studies discussed above and our work is that while they provide data recoverability, they ignored the integrity protection of data and counters through message authentication code (MAC) and Bonsai Merkle Tree (BMT), respectively. The security of any counter-mode encryption techniques is predicated upon avoiding counter reuse [21] which is provided through BMT. Consequently, in those studies, security guarantees are not preserved as the system is vulnerable to direct tampering of data and counter replay attack [22]. Rogers et al. [37] address the issue of implementing integrity-verification and encryption in shared multi-processor systems. Similar work also addressed the issue of shared multi-processor systems by providing more efficient protection for processor-processor communication and coherence messages [38]. To our knowledge, our work is the first to take a holistic approach of including all security metadata and discuss issues pertaining their persistence, performance, resilience, and recovery time. Furthermore, unique to our study is the investigation of how persistent and non-persistent regions of memory must be treated differently.

Persisting data in NVMM have different models based on the required software and hardware changes [13, 39–41]. Several APIs and libraries have been developed for the purpose of persisting data in NVMM and utilizing the OS support for PMEM, e.g., Intel PMDK [24]. In this paper, our support focuses on persisting security metadata accompanying persistent data when being written to NVMM, and thus we do not require any changes at the application or library level, hence Triad-NVM can be integrated with most persistency models.

Finally, other studies that target reducing NVMM writes, such as [9, 10], have not considered the problem of persisting security metadata.

7 CONCLUSION

In this paper, we proposed Triad-NVM, an architecture that provides secure persistent memory, where we discussed for the first time how persistency of data and security metadata (counters, MACs, and Merkle Tree) must be provided for both persistent and non-persistent memory regions of non-volatile main memory. We explored holistically issues of persistence, performance, resilience, and recovery time and show how our proposed optimizations affect said issues. Our evaluation results show that Triad-NVM improves throughput by roughly 2× compared to strict persistence, while achieving recovery time of less than 4 seconds for an 8TB NVMM system, which is more than three orders of magnitude smaller than a system without security metadata persistence.

ACKNOWLEDGEMENT

Much of this work was done when Amro Awad was a summer faculty fellow at the U.S. Air Force Research Laboratory funded through the AFOSR SFFP program. Mao Ye and Kazi Abu Zubair were funded through UCF. Yan Solihin was supported in part by UCF. We thank the anonymous reviewers for their feedback.

REFERENCES

[1] "Enhancing High-Performance Computing with Persistent Memory Technology." https://software.intel.com/en-us/articles/enhancing-high-performance-computing-with-persistent-memory-technology. Accessed: 2018-12-07.

[2] "Reimagining the Data Center Memory and Storage Hierarchy." https://newsroom.intel.com/editorials/re-architecting-data-center-memory-storage-hierarchy/. Accessed: 2018-12-07.

[3] S. Chhabra and Y. Solihin, "i-nvmm: a secure non-volatile main memory system with incremental encryption," in *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, pp. 177–188, IEEE, 2011.

[4] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[5] R. Geambasu, T. Kohno, A. A. Levy, and H. M. Levy, "Vanish: Increasing data privacy with self-destructing data.," in *USENIX Security Symposium*, vol. 316, 2009.

[6] T. Pettersson, "Cryptographic key recovery from linux memory dumps," 2007.

[7] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," in *ACM SIGARCH Computer Architecture News*, vol. 34, pp. 179–190, IEEE Computer Society, 2006.

[8] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, and M. K. Qureshi, "Synergy: Rethinking secure-memory design for error-correcting memories," in *The 24th International Symposium on High-Performance Computer Architecture (HPCA)*, 2018.

[9] A. Awad, P. Manadhata, S. Haber, Y. Solihin, and W. Horne, "Silent shredder: Zero-cost shredding for secure non-volatile main memory controllers," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, (New York, NY, USA), pp. 263–276, ACM, 2016.

[10] V. Young, P. J. Nair, and M. K. Qureshi, "Deuce: Write-efficient encryption for non-volatile memories," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, (New York, NY, USA), pp. 33–44, ACM, 2015.

[11] J. Rakshit and K. Mohanram, "Assure: Authentication scheme for secure energy efficient non-volatile memories," in *Proceedings of the 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, June 2017.

[12] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via justdo logging," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 427–442, 2016.

[13] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "Thynvm: Enabling software-transparent crash consistency in persistent memory systems," in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 672–685, ACM, 2015.

[14] J. Arulraj and A. Pavlo, "How to build a non-volatile memory database management system," in *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 1753–1758, ACM, 2017.

[15] T. Wang and R. Johnson, "Scalable logging through emerging non-volatile memory," *Proceedings of the VLDB Endowment*, vol. 7, no. 10, pp. 865–876, 2014.

[16] J. Arulraj, A. Pavlo, and S. R. Dulloor, "Let's talk about storage & recovery methods for non-volatile memory database systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 707–722, ACM, 2015.

[17] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won, "Nvwal: Exploiting nvram in write-ahead logging," *ACM SIGOPS Operating Systems Review*, vol. 50, no. 2, pp. 385–398, 2016.

[18] S. Liu, A. Kolli, J. Ren, and S. Khan, "Crash consistency in encrypted non-volatile main memory systems," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 310–323, Feb 2018.

[19] M. Ye, C. Hughes, and A. Awad, "Osiris: A Low-Overhead Mechanism for Restoration of Secure Non-Volatile Memories," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[20] P. Zuo and Y. Hua, "Secpm: a secure and persistent memory system for non-volatile memory," in *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, (Boston, MA), USENIX Association, 2018.

[21] A. S. Tanenbaum, *Computer Networks*. New Jersey: Pearson Education, Inc., 4 ed., 2003.

[22] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 183–196, IEEE Computer Society, 2007.

[23] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.

[24] U. Usharani, "Introduction to Programming with Persistent Memory from Intel."

[25] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, pp. 1–17, Sept. 2006.

[26] Micron, "Breakthrough nonvolatile memory technology."

[27] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *IEEE micro*, no. 1, pp. 143–143, 2010.

[28] B. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, 2009.

[29] Z. Li, R. Zhou, and T. Li, "Exploring high-performance and energy proportional interface for phase change memory systems," *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 210–221, 2013.

[30] "Linux Direct Access of Files (DAX)."

[31] A. Awad, Y. Wang, D. Shands, and Y. Solihin, "Obfusmem: A low-overhead access obfuscation for trusted memories," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 107–119, ACM, 2017.

[32] S. V. A. Joshi, V. Nagarajan and M. Cintra, "Atom: Atomic durability in non-volatile memory through hardware logging," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2017.

[33] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for nvm," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.

[34] A. M. Rudoff, "Deprecating the pcommit instruction," 2016.

[35] T. Holterbach, S. Vissicchio, A. Dainotti, and L. Vanbever, "Swift: Predictive fast reroute," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pp. 460–473, ACM, 2017.

[36] K. Abu Zubair and A. Awad, "Anubis: Low-overhead and practical recovery time for secure non-volatile memories," in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, 2019.

[37] B. Rogers, C. Yan, S. Chhabra, M. Prvulovic, and Y. Solihin, "Single-level integrity and confidentiality protection for distributed shared memory multiprocessors," in *Proceedings of the IEEE 14th International Symposium on High Performance Computer Architecture*, pp. 161–172, IEEE, 2008.

[38] B. Rogers, M. Prvulovic, and Y. Solihin, "Efficient data protection for distributed shared memory multiprocessors," in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pp. 84–94, ACM, 2006.

[39] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories," *ACM Sigplan Notices*, vol. 46, no. 3, pp. 105–118, 2011.

[40] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language-level persistency," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 481–493, ACM, 2017.

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