2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-7652-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/HPCA56546.2023.10070991

Thoth: Bridging the Gap Between Persistently Secure Memories and Memory Interfaces of Emerging NVMs

Xijing Han
North Carolina State University
Raleigh, USA
xhan24@ncsu.edu

James Tuck
North Carolina State University
Raleigh, USA
jtuck@ncsu.edu

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

***Abstract*—Emerging non-volatile memories (NVMs) are expected to be part of future computing systems, including cloud systems and edge devices. In addition to the high density (and hence large capacities) NVMs can provide, they feature ultra-low idle power which makes them very promising for edge computing and data centers. Additionally, NVMs' ability to retain data upon system crash (e.g., power outage or software bug) makes them a great candidate for high-availability and persistent applications. However, NVMs' data retention capability brings in security challenges and further complicates today's secure memory implementations; to ensure correct and secure system recovery, the data and security metadata must be persisted atomically (i.e., up-to-date in memory upon a crash).**

Despite the many efforts for rethinking secure memory implementations to enable crash-consistency, we observe that the state-of-the-art solutions are based on a major assumption that may not be suitable for future memory interfaces. Specifically, the majority of today's solutions assume that either the encryption counter and/or message-authentication code (MAC) can be co-located with data by directly or indirectly leveraging the otherwise Error-Correcting Codes (ECC) bits. However, we observe that emerging interfaces and standards delegate the ECC calculation and management to happen inside the memory module, which makes it possible to remove extra bits for ECC in memory interfaces. Thus, all today's solutions may need to separately persist the encrypted data, its MAC, and its encryption counter upon each memory write. To mitigate this issue, we propose a novel solution, Thoth, which leverages a novel off-chip persistent partial updates combine buffer that can ensure crash consistency at the cost of a fraction of the write amplification by the state-of-the-art solutions when adapted to future interfaces. Based on our evaluation, Thoth improves the performance by an average of 1.22x (up to 1.44x) while reducing write traffic by an average of 32% (up to 40%) compared to the baseline Anubis when adapted to future interfaces.

***Index Terms*—Persistent Memory; Security Metadata; Secure NVM**

I. Introduction

Emerging non-volatile memories (NVMs) are promising technologies to augment/replace DRAM in future computing systems [20], [25], [34], [48]. Although they are still in their infancy stage, emerging NVMs (e.g., Intel's DCPMM) are already getting deployed in major cloud systems [31], and this is expected to grow over time [40]. For instance, SK Hynix has recently announced its 3DVXP technology [45] which promises a better scalability than Intel's and Micron's 3DXPoint[1]. In addition to their high density (hence high capacity modules), they offer ultra-low idle power which offers great energy savings for data centers compared to DRAM; DRAM requires periodic refresh operation for each memory cell and hence its power consumption increases with its capacity [32]. In today's servers, around 40% of the energy consumption is attributed to DRAM refresh operations [8], [39]. Finally, in addition to NVMs' capacity and power advantages, they can retain data even in the events of power outage. Such data persistence capability can be leveraged to enable hosting persistent files in NVM memory modules [27], [41] and facilitate crash-consistent applications [22]. On the other hand, NVMs have limited write endurance [7], [34], [46] and facilitate data remanence attacks [13], [18].

With security becoming a first-class citizen design requirement for computing systems, most processor vendors are racing into providing security primitives that enable a safe execution environment [2], [4], [5], [12], [14], [26]. While the threat model and hence security guarantees, implementation details, and maturity of these supports vary significantly, a common theme between all of them is the need for protecting data confidentiality and integrity when leaving the processor chip. While efficient memory integrity and confidentiality protection have reached to an acceptable maturity level for conventional memory (i.e., DRAM) [11], [17], [35], recent studies have shown that today's secure memory implementations are incompatible with NVMs [16], [19], [28], [29], [44], [49]. Such incompatibility mainly arises due to crash consistency issues between data and its corresponding security metadata (e.g., encryption counter and authentication code) [7], [28], [44]; once the encrypted data reaches the persistence domain, it needs to also have its security metadata in the persistence domain as well. Otherwise, the encrypted data cannot be decrypted or authenticated upon recovery from a crash. Meanwhile, naively persisting security metadata along

[1]Although Intel has recently abandoned its memory and NAND flash business to focus on other market sectors, emerging NVMs remain to be major investment plans for key memory companies such as SK Hynix, Samsung and Toshiba.

with data on each memory write can lead to significant performance degradation and lifetime reduction of NVMs [6], [10], [28], [29], [43], [44], [49].

To ensure crash consistency of secure NVMs while incurring minimal write and performance overheads, state-of-the-art solutions [10], [19], [42]–[44], [49] rely on persisting (part of) the security metadata atomically with data through repurposing Error-Correcting Codes (ECCs) that are co-located with the data. For instance, recent secure NVM works [3], [10], [10], [42], [49] commonly assume that ECC codes can be encrypted and overridden in a way that enables recovering the encryption after limited number of trials, as proposed in Osiris [44]. Other works proposed to override ECC with Message Authentication Code (MAC) [43], as done in secure DRAM for performance reasons [36]. In all these works, the majority of the write reduction is based on leveraging the ECC bits (typically 64-bits) written atomically along with the data. Most importantly, the fact that these bits are stored along with the data, and hence can be used to store security metadata (encryption counter, MAC, or ECC/counter [44]). However, if such extra bits for ECC are not needed in future memory interface, the security metadata need to be persisted as separate writes. Specifically, ***if future memory interfaces do not have extra bits that are suitable for co-locating secure metadata with data, then there are no effective solutions for persistently secure NVMs. The only available solution is to incur separate security metadata writes with each persistent memory write in an atomic fashion***.

The Problem: Emerging NVM modules, such as Intel's DCPMM modules, compute ECC bits internally and attempt to correct them before reporting errors (interrupt) to the host [1], [24]. Such DIMM internal implementation of ECC is likely to dominate the memory industry due to the following: (1) many memory technologies with different reliability (and hence ECC algorithms) are available, and thus relying on the processor-side memory controller to implement them is infeasible. (2) Leaving internal ECC implementation details to memory vendors allows more flexibility in choosing suitable ECC memory for critical domains (e.g., safety-critical systems) while more relaxed (or absence thereof) ECC in less critical domains. (3) With certain modules supporting encryption internally, the best place to calculate ECC is after the encryption is done, otherwise the errors diffuse after decryption and makes it difficult for host-level ECC to fix it. For instance, Intel's DCPMM internally leverages AES-XTS and hence if there is no strong ECC support inside the module then errors are further exacerbated when decrypted internally before being sent to the host[2]. Moreover, the granularity of AES-XTS cipher blocks is 128 bits and hence even a single bit error within a 128-bit of the memory block can turn a 128-bit into a nearly-random value after decryption inside the DIMM; making it challenging to fix at the host side even with strong ECC support. Similarly, future NVMs (e.g., SK Hynix's 3DVXP) are envisioned to be interfaced through computer express link (CXL) memory semantic protocol [37], where the width is 66B of which only 2B are used for ECC of the transmission, and the remaining 64B is the payload (e.g., cacheline).

The Challenge: In today's state-of-the-art secure NVM implementations [10], [19], [36], [42]–[44], [49], the message-authentication codes (MACs) of data is assumed co-located with data using additional pins, while the encryption counter used to encrypt the data is persisted through overriding (or repurposing) the ECC bits. Without the need for host-side ECC, which could be replaced by on-DIMM ECC in future interfaces, such implementations may lose their effectiveness. Specifically, they will require two additional writes for the MAC and counter blocks. Such overheads are unacceptable in terms of both NVM lifetime and performance. Due to storage efficiency, encryption counters and MACs are not co-located in the same memory block; encryption counters has much less storage overhead compared to MACs (typically 12.5% for MACs vs. 1.56% for counters), and hence they are separated in different blocks, which causes two extra separate block writes to memory for each memory block write.

The Solution: Our key observation is that much of the write amplification in secure NVM occurs because of the disparity between counter and MAC sizes and the write granularity to memory. On a memory write, an additional 8B MAC and 8B counter (minor + major counters [11]) are updated along with the data, and they are held in separate blocks in memory. Because only one counter or MAC in each block is updated, we refer to these as a partial updates. When partial updates are persisted to memory to maintain crash consistency, both full blocks (of the counter and MAC blocks) need to be persisted causing the write amplification[3].

Our main insight is that we can pair a large off-chip persistent buffer for partial updates with the normal write-back behavior of our secure metadata cache to create a new write efficient architecture. Partial updates are packed together efficiently and written into the persistent buffer to provide crash consistency while the secure metadata cache continues operate as a write-back cache. We observe that if we buffer partial updates for long enough, when they are evicted from the persistent buffer, no additional writes are needed because the state they hold has already been written to memory efficiently through other means. For example, the secure metadata cache will eventually perform the update through its natural write-back process, likely after accumulating many updates to the same entry saving many writes. Another example is if another yet younger partial update is made for the same metadata (i.e., MAC or counter), then all older partial updates to the same metadata are stale and can be discarded, saving those writes. Moreover, if the metadata cache block is persisted for any reason, then all partial updates for that same metadata block can be safely discarded because the metadata block in memory

[2]Unlike AES-CTR mode which is generally used in the processor side for confidential computing, AES-XTS directly feeds the ciphertext as input to the AES algorithm to complete decryption, and hence significantly diffuses any bit errors upon decryption.

[3]The write amplification further grows with NVMs that use larger access granularity (128B or 256B in Intel's DCPMM [38]) while the MAC and counter sizes remain the same.

is already up-to-date. With these insights, we conclude that a large persistent buffer can provide lots of opportunities to avoid writing partial updates to memory. Most importantly, we observe that the overheads for maintaining such a persistence buffer are minimal because we do not need the persistent buffer to reside on-chip. Many partial updates can be packed into one memory block and written to memory together. Thus, with relatively minimal memory buffering overheads, we significantly increase the probability of eliminating the full-block write upon each partial update.

In this paper, we present our proposed solution, ***Thoth***, which ensures crash-consistency while exploiting temporal and spatial combining of partial updates in a secure and elegant manner. Thoth aims to (1) realize a persistent ***partial updates buffer (PUB)*** in memory (2) upon eviction of a partial update entry from the PUB, discard the write-back of the corresponding metadata block when no longer necessary (i.e., updated later or evicted from metadata cache in processor). To evaluate Thoth, we use Gem5 [9], a full-system cycle-level simulator with five representative persistent applications. Based on our evaluation, Thoth improves the performance by 1.22x on average (up to 1.44x), compared to the state-of-the-art solution Anubis [49] when adapted to work with modern memory interfaces (i.e., persist MAC and counter blocks upon each write) and augmented with a small on-chip persistent write queue as in Intel's Write-Pending Queue (WPQ). Furthermore, Thoth improves the NVM lifetime by reducing the number of writes to 32% the Anubis baseline. Thus, we conclude that carefully-handled large partial persistent buffers can have great potential for reducing security metadata persistence overheads in future memory interfaces.

The rest of the paper is organized as follows. First, in Section II, we present the background and the main concepts related to Thoth. Later, in Section III, we quantitatively demonstrate the potential for PUB and its effectiveness in eliminating much of the otherwise-occurring full-block security metadata writes. Section IV presents the design space discussion and our proposed design of Thoth. Our methodology, evaluation and analysis are presented in Section V. Section VI presents the related work, and Section VII concludes.

II. Background

A. Memory Encryption & Integrity Protection

In secure memory, both the *confidentiality* and the *integrity* of the memory data are protected [6], [11], [29], [35], [49].

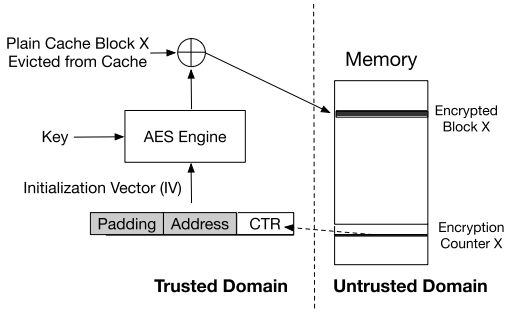

Fig. 1. Counter-mode Memory Encryption Mode.

Confidentiality: the data confidentiality is protected through encrypting memory blocks once leave the processor chip and decrypting them when read back. Generally speaking, there are two ways commonly used to implement memory encryption: *direct encryption* and *counter-mode encryption*. Direct encryption, also known as electronic code book (ECB), feeds the data to be encrypted/decrypted directly as an input to the encryption algorithm (e.g., AES or DES). Meanwhile, counter-mode encryption uses an initialization vector (IV) as the input of the encryption algorithm to generate a one-time encryption pad (OTP), where the OTP will be simply XOR'ed with the plaintext/ciphertext to complete the encryption decryption process. Due to its ability to hide the latency of the encryption algorithm (fetching data can be overlapped with generating OTP), and its ability to provide spatial and temporal uniqueness of ciphertexts for the same plaintext, it is the commonly used as the default memory encryption scheme [7], [11], [35], [44], [49]. Figure 1 depicts counter-mode encryption at high-level.

As depicted in Figure 1, counter-mode memory encryption relies on counters that each is associated with a single memory block (the access granularity to memory). Upon decryption/encryption, the counter corresponds to the block of interest is fetched from memory and used along with block address (to ensure spatial uniqueness for similar data written to different locations) and padding, all forming the initialization vector (IV) to generate the encryption/decryption pad. Note that the counter provides temporal uniqueness of the ciphertext when the same value is written again to the same location; it gets incremented each time the corresponding block is encrypted and written to memory. Counters must be sized enough such that they do not overflow, as this can lead to major attacks such as known-plaintext attacks and data replay. However, this can lead to large storage overheads. Thus, state-of-the-art counter organization schemes use a split-counter scheme mode [11] where a counter is comprised of a 64-bit major part that is shared across all blocks of a page, and a 7-bit minor specific for each block. Thus, a 64B counter block can fit minor counters of 64 memory blocks and along with their common major counter [11]. Also, counter-mode encryption requires the integrity of counters to be protected. Due to total size of counters, they are stored off-chip and only cached on-chip upon access, similar to data. For the rest of the paper, we use counter-mode encryption and split-counter mode.

Integrity: for memory integrity, Merkle Tree (MT) is generally used where the root of the tree is always kept inside the processor chip [35]. MT can be thought of as a tree of hashes, where each level is the hashes of its immediate children level nodes. To speed-up update and verification operations of the MT, a volatile cache inside the processor chip is used to cache the most recent access tree nodes. Due to their high spatial locality, integrity tree caches generally have very high hit rates. To minimize the storage overheads of MT, prior work [35] makes the observation that it is sufficient to build integrity tree of encryption counters while protecting the encrypted data through MACs calculated over the ciphertext, address, and

counter; since the counter's freshness is guaranteed through the integrity tree, the ciphertext is guaranteed to be fresh since the MAC is also calculated over the counter. In other words, no need to build an integrity tree over data to ensure its freshness, but it is sufficient to rely on MACs that are calculated over the data and its counter. Such a scheme is called *Bonsai Merkle Tree (BMT)* [35], and it is commonly used in most recent works. Although our work is orthogonal to the integrity tree scheme and implementation, and can be used with tree scheme, we assume BMT for the rest of the paper due to its wide use in secure memory [6], [10], [19], [42], [44], [49], [50].

B. Persistently-Secure Memories

One of the main features of emerging NVMs is the ability to retain data upon system crash and power-failure events. However, to enable that, the data and its corresponding security metadata must be persisted atomically. Alternatively, there should be a mechanism to recover/infer such security metadata in a secure fashion. The BMT can be inferred, reproduced and verified through the root [6], [49]. The BMT recovery process can be sped-up using low-overhead tracking as shown in prior work [49]. However, the encryption counter and data MAC must be recovered and cannot be inferred from other elements. For instance, the MAC uses the counter however the MAC is also calculated over the data, and thus the most-recent MAC must be persistent along with the data to verify its integrity.

Prior works leverage ECC bits written with data to store MAC [36], [43] or (part of) encryption counter [6], [10], [19], [42], [44], [49]. For instance, in Osiris [44], the encryption counter is embedded in the ECC bits through encrypting ECC along with the plaintext. Upon decryption, if a wrong/stale counter is used, the ECC will be unrelated (large number of code words will indicate error with high probability), and thus a stale counter can be identified. By limiting the divergence between the persisted counter in memory and the counter used for encryption, a limited number of trial is used until the counter used for encryption is recovered for each block. For MACs, Osiris assumes an independent MAC chip that can also be written concurrently with data. Most state-of-the-art solutions in secure NVM [6], [10], [19], [42]–[44], [47], [49], [50] builds on the idea of co-locating such security metadata with data to minimize the write amplification for implementing crash-consistent secure NVMs. In this paper, we address the problem of crash-consistent secure memory with a more generic and realistic emerging memory interfaces where no host-accessible ECC are available as in DDR-T [1], [24], CXL-based modules (e.g., SK Hynix 3DVXP [45]), and DDR5 modules in systems where the DDR5 on-die ECC support is considered sufficient [30], i.e., no host-managed ECC are used.

Asynchronous DRAM Refresh (ADR): persistent memory standards require extra residual power to flush contents that might be still in the volatile domain upon a crash. The mandatory ADR support requires sufficient backup-power (e.g., through ultra-capacitor) to flush a small buffer inside the memory controller, called write-pending queue (WPQ). Hence persistent applications do not need to wait until data is written to NVM. On the other hand, there is an optional feature called ***enhanced ADR (eADR)*** which allows flushing the whole cache hierarchy. Unfortunately, eADR support is limited to certain power supplies, and hence disabled even in recent servers [23], and can lead to complexity in maintenance, and increase in cost and carbon footprint for data centers [15], especially with recent large cache hierarchies (e.g., AMD's 512MB V-Cache in EPYC processors). Accordingly, we assume basic ADR support in our system, and leave investigating systems with optional eADR support to future work.

C. On-Die ECC in Emerging Memory Technologies and Interfaces

For current NVM offerings used as main memory, e.g., Intel's DCPMM, a proprietary interface (DDR-T [24]) is used where also ECC is calculated through ECC engine inside the DIMM [1], [24]. We also observe that similar trend for on-DIMM ECC capabilities started to become part of the DDR5 standard for DRAM [30]. Hence, even for DRAM, can now reliably deploy large size memories without the need for stringent reliability support from the host. While for DDR5, there is an option to deploy additional host-level ECC for stringent reliability requirements, the availability of ECC on-DIMM by default will enable more users to take advantage of larger memory reliably at lower cost. Finally, as discussed earlier, when internal encryption is supported as in Intel's DCPMM, there should be a strong ECC support on the DIMM side to enable correction before decryption, otherwise errors can diffuse and hence potentially defeat the purpose of any host-level ECC support. Simiarly, future CXL-based NVMs are envisioned to have their own internal ECC implementations given the limited fixed width (66B) of CXL.

Accordingly, prior works in secure NVM are no longer effective as they require two extra persistent writes (MAC and counter blocks) for each persistent memory write if host-side ECC can be replaced with On-Die ECC. To enable secure processors with emerging memory technologies and interfaces, we need an efficient mechanism to minimize such writes under these constraints while ensuring crash consistency.

III. Motivation

In this section, we demonstrate the major observations we have about the effectiveness of long-term buffering of partial updates, which will later motivate our design decisions for Thoth. First, we define *partial security metadata block updates* as updates to a specific MAC or encryption counter within a MAC cache block or counter block. Such partial updates occur due to writing the memory block protected by these MAC and counter. However, due to the access granularity in memory, MACs, and similarly counters, are grouped into memory blocks that are fetched and cached together (e.g., in 64B block granularity for conventional DDR). Thus, upon an update to a MAC or a counter block, the whole blocks that contain them will be persisted to NVM in addition to the data block. Whole block persists triggered by partial security metadata block updates are key source of write amplification

for secure NVMs and we want to avoid as many as possible to enable greater compatibility with emerging memory interfaces.

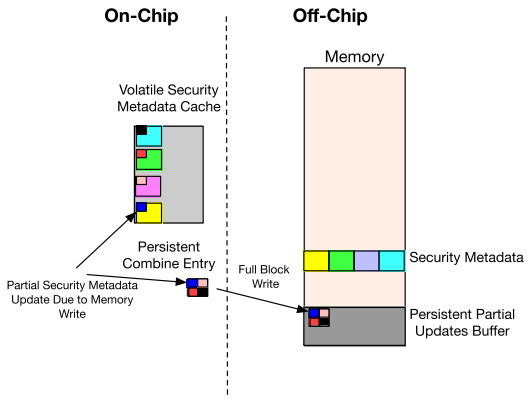

Fig. 2. High-level example of partial updates buffer (PUB) system-level layout.

We imagine a new organization for crash consistent NVM that is able to efficiently reduce write amplification. As shown in Figure 2, we add a partial updates buffer (PUB) in memory to collect these partial updates. As shown in the figure, we still use a write-back security metadata cache that is updated on each security metadata update. However instead of also persisting the full metadata block to memory due to a partial update, we combine these partial updates, pack them tightly into blocks, and buffer them in memory at a fraction of the number of writes they would otherwise cost. Note that during run-time, all metadata will be strictly fetched and evicted through the normal path and not through the PUB, e.g. we check for it in the cache, and otherwise bring it from its original location (not the PUB). Also, upon eviction from the secure metadata cache, if dirty, then it needs to be written to its original location. However, upon crash events, the most recent update in the volatile secure metadata cache might be lost, and thus we need to recover the partial updates from the PUB. This architecture may at first appear to increase the number of writes to memory since evictions of the partial updates from the PUB would ultimately require full block persists, however, this organization actually significantly reduces writes.

Our ***key observation*** that enables this reduction in writes is that the vast majority of partial security metadata updates when evicted from the PUB need no additional writes, if they are persistently buffered for long enough. This is because, after a long enough time passes, the probability is high that memory already contains their update. The following reasons explain why the probability is so high. (1) While the partial update is buffered, the security metadata block which the partial update belongs to may have already been evicted from the secure metadata cache and persisted to NVM; such a probability increases over time due to the natural changes in the working set of applications over time. (2) An older partial update eviction may have already caused a full security metadata block persist which included the partial update from the current entry (already in the persisted cached metadata block). Note, this case captures spatial locality of updates to metadata within the same block. (3) A younger partial update arrived to the same location as an existing partial update. In this case, any older partial update for the same metadata can be discarded because it is stale. This case captures the temporal locality associated with a data block that is frequently updated, a common trait in persistent applications.

We observe that the aforementioned three cases are very common and their probabilities increase significantly with the increase of the partial buffer size. Figure 3 breaks down the cases we observe upon eviction from a hypothetical FIFO partial buffers with sizes of 500,000 entries (A), 5,000 entries (B), and 50 entries (C). The ***written-back*** percentage represents the probability of upon the eviction of a partial update its security metadata cache block still needs to be persisted to memory. Meanwhile, ***already-evicted*** represents the case of when upon a partial update eviction from the buffer its up-to-date metadata block has been already evicted from cache and written back to memory, and thus the partial update can be discarded safely. The ***clean copy*** case is when a partial update is upon eviction, its metadata block is still in the metadata cache but in clean state, which means it was either persisted due to partial update eviction or evicted from cache then fetched again later. Finally, the ***stale copy*** indicates the case where upon the eviction of a partial update its metadata block was in the cache and dirty, however the partial update is stale, i.e., a newer partial update was inserted in the PUB. Thus, the stale partial update can be safely discarded.

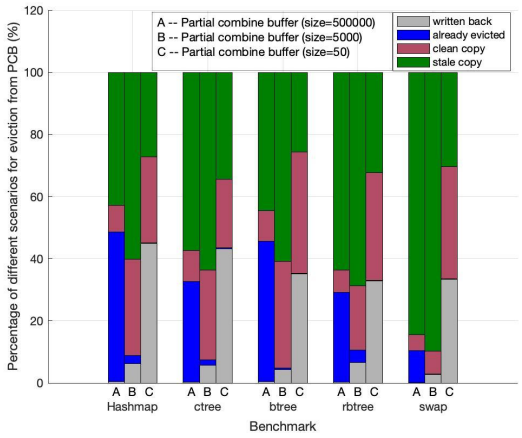

Fig. 3. Breakdown different scenarios for evicted copies from Partial Combine Buffer (PCB).

As shown in Figure 3, for all the benchmarks we can observe that with a large enough partial buffer size, the majority of partial updates (99.5% on average for the 500,000 buffer) do not cause a full block persist upon eviction. Rather when the larger PUB sizes are used, the common case is that evicted entries are stale, and the second most common case is that the partial update has already been evicted and written back from the secure metadata cache. In neither case is it necessary to write the evicted PUB entry back to its original location in memory to ensure crash recoverability.

Based on these insights, we observe the importance of

large PUB and their great potential for eliminating extra security metadata block persistence operations that ensure crash consistency. In short, our approach replaces the certain extra security metadata writes with a very small probability of an extra write plus the low overhead of buffering packed partial updates off-chip. The rest of the paper demonstrates how to realize this hypothetical design in an elegant yet highly effective fashion.

IV. Design

In this section, we describe the design options and rationale behind the various design choices for Thoth. First, we define our threat model.

Threat model: Our threat model is similar to the state-of-the-art secure NVM work [10], [19], [29], [43], [44], [49], [50] in all aspects: bus-snooping attacks, physical thefts, memory scanning, memory replay, and potential data tampering. In addition to the conventional secure DRAM guarantees [11], [35], [36], secure NVMs also need to ensure crash consistency and the security of data across data reboot episodes. Similar to prior work in secure NVMs, side-channel attacks such as power-side channel attacks, memory timing attacks, and access pattern leakage are beyond the scope of this work. Such types of attacks have a wide range of solutions in literature that can be adopted directly.

First, let's more formally define the ***partial updates buffer (PUB)*** alluded to earlier in Section III. A PUB contains partial security cache block updates, i.e., only the portions to be updated for security metadata blocks. For instance, a memory write would cause updating the MAC block containing the MAC and the counter block containing the counter correspond to the memory location to be written. However, the partial updates are the MAC (8B field) and the impacted counter (typically just the 7-bit minor counter). Thus, a PUB should merely contain the new values of the impacted MAC and (minor) counter. To ensure correctness we need to ensure the following: ① the partial updates need to be securely recoverable and protected. ② the volatile security metadata cache should be updated with the most recent metadata (e.g., MACs and counters) upon buffering partial updates to the PUB; this ensures consistency. ③ there should be a safe handling eviction policy when the PUB fills up.

As shown earlier, in Section III, the PUB needs to be quite large (e.g., 500,000 entries) to maximize the benefits. Thus, the logical place to place the PUB is off-chip. However, this brings us to our first design question, how do we arrange partial updates in memory?

A. PUB Organization

Since the memory write granularity is at a cache block granularity (e.g., 128B or 256B), we need to *persistently* pack partial updates into blocks before writing them back to memory. The buffer itself is managed as a FIFO circular buffer where two counters are used, one to indicate the start and one to indicate the end. A third register is used to indicate the base address of the buffer. Once the start equals the end, no more insertions are allowed until evictions occur (and hence the end variable is incremented mod the buffer size).

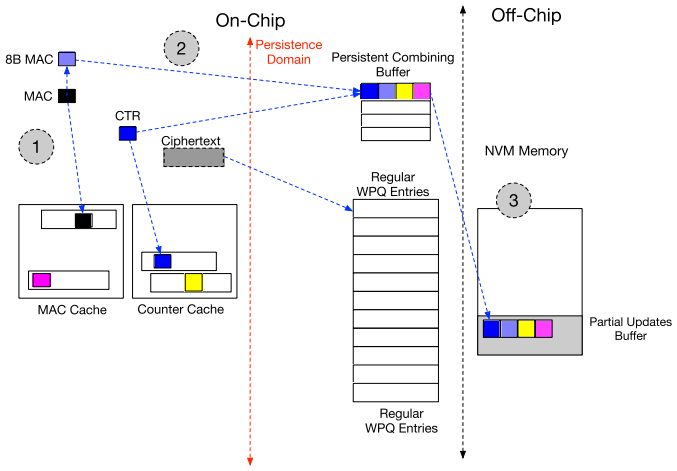

Fig. 4. Overview of the coalescing steps leveraging reserved WPQ entries.

To persistently hold partial updates as they are forming a full cache block to be written off-chip to the PUB, a few entries are reserved from the processor's internal WPQ which is battery-backed through the ADR support [21].

As shown in Figure 4, multiple partial updates can be combined together in a single WPQ entry. When a new memory write occurs, after obtaining the verified counter value and calculating the new MACs, calculating the new integrity tree root and nodes[4], the new ciphertext and updated security metadata need to be persisted before the transaction is considered complete and persistent. Since an 8-to-1 MAC is computed over the ciphertext, a 16B MAC is computed for 128B cache block and a 32B MAC for 256B. To be able to pack more partial updates in one WPQ entry, an 8B second-level MAC is computed and allocated in PUB. As shown in Step ①, a specific ciphertext and its accompanying security metadata (MACs and counter) are ready to be persisted, and the counter and first-level MAC can be made to caches. In Step ②, which can be overlapped with ①, the partial updates (counter and 8B second-level MAC, not their full cache blocks) are placed in a reserved WPQ entry, while the ciphertext block is inserted in the regular WPQ. Now as the reserved coalescing WPQ entry is full, it can be written to the PUB as shown in Step ③. We omitted the steps for updating the bounds of the buffer upon insertion for the sake of simplicity.

Each PUB cache block is comprised of 9 partial updates (for 128B cacheline size) or 19 partial updates (for 256B cacheline size). A partial update entry contains the {address, MAC, counter, status}. The *address* is 32b and represents the cache block address which the MAC and counter correspond to, this can address address up to a 512GB module (note that there are unused bits that can be used for address if larger modules are used in the future). The *counter* is the 7b minor counter; we persist the counter block immediately when a minor counter

[4]Note that this also can be done lazily by securely tracking the cache content as done in Anubis [49] and Phoenix [3].

overflow occurs (i.e., the major counter changes). The MAC is 64b for both designs with 128B cache blocks and 256B cache blocks. Finally, the *status* bits (2b) are used to help on deciding the actions upon the eviction of this partial update entry from the PUB (will be discussed later in this section). To eliminate issues with crashes while the coalescing entry in WPQ is not full yet, we duplicate the existing partial entries upon a crash to fill a full cache block.

B. PUB Eviction Mechanism

The second question we aim to answer is: how to efficiently implement an eviction policy from the PUB? Note that this is perhaps the most critical aspect in Thoth; without an efficient mechanism, most partial updates will end up eventually causing their corresponding metadata blocks to be persisted when these partial update entries are evicted from the buffer. Upon an eviction of an entry from the PUB, it is time to decide whether to persist the partial security metadata blocks' updates to their original locations or not. To implement this, upon an eviction from the PUB entry, we fetch the victim (e.g., last) partial updates memory block from the PUB, which contains a pack of updates. For each entry, we check to see if we actually still need to persist that partial update or not, i.e., if it has been already persisted through the natural cache writeback path, persisted due to a later partial update to the exact same MAC/CTR, or simply a prior PUB entry that falls within the same security metadata block has been evicted, and hence caused the spatially-shared security metadata block in cache to be persisted.

Security Aspects: The first aspect that arises here is whether or not we need to verify the integrity of the partial updates entries once they are read back for eviction from the PUB (and potentially persist their corresponding security metadata). Since the most recent counter/MAC values are also either in the volatile counter/MAC cache or already evicted to memory, the partial update value is never needed or incorporated in any update during run-time and normal operation. In other words, during normal execution time, original location of secure metadata blocks in NVM is always updated by the copies in secure metadata cache. However, upon a crash, the most recent values of these updates in the cache are lost, and thus we need to recover them through reading the PUB. Fortunately, since the most recent counter values are already incorporated in the integrity tree root persistent inside the processor [6], or a secure and protected shadow cache in memory (as in [49]), we can verify these partial updates upon secure reconstruction. In other words, during crash, tampered PUB is detected using the integrity trees. For example, tampering a most-recent counter value that is lost in the cache is detected by leveraging the root calculated over secure metadata cache. Tampering a counter value that is already updated in-place before crash will result in failure in reconstruction of the main integrity tree. Thoth only replaces encoding MAC/CTR blocks using extra bits at the memory bus with a temporary combination buffer before they end up being updated in place.

Detection of Stale Partial Updates: Another major aspect is how to efficiently detect if a partial update is no longer needed, e.g. due to the cache eviction of the security metadata cache block already containing this update to memory. As mentioned earlier, we aim to identify the following three cases: ① the up-to-date cache block containing the partial update has been already evicted from cache. Note that this case should also incorporate the scenario for evicting the cache block after the partial update occurred, but it has been read later and other parts of the block were updated. ② the same partial metadata has been updated later and added to the partial buffer ③ a prior metadata block persist operation, e.g., due to eviction of a PUB entry, caused other partial updates that share the security metadata block to be already persisted along with the block. To allow detecting these cases, we start with a simple design, which we call **Write-Back Through Bitmask Checks (WTBC)**, that relies on fine-granularity tracking of dirtiness of counters/MACs within security metadata blocks. Upon a fetch of a security metadata block, all dirty bits for all its MAC/CTRs are set to 0. Only upon a partial update to any of them will that specific MAC/CTR's corresponding dirty bit be set to 1. Also, upon persisting a metadata block, all of its dirty bits are reset to 0. Figure 5 depicts scenarios that reflect how WTBC captures cases ① and ③, in Event 4 and Event 6, respectively. Unfortunately, merely relying on the dirty bits to capture case ② is insufficient; if any write to the same data block occurs between the evictions of two prior partial updates correspond to the same data block, then the later eviction would still cause a persist of the metadata block as the dirty bit will be one. However, the first partial update eviction could have already caused persistence of the block which potentially contained the later partial update. One more efficient way to capture case ② is to compare the value to be evicted from the PUB with the verified value in the metadata cache. Based on our observation that each partial update for MAC/CTR will generate a new unique value; thus, by comparing that value along with the dirty bit value, we can know if the partial update is stale or not.(Note, evicted partial update's MAC needs to be compared with a second level 8B MAC computed over the corresponding MAC in the secure metadata cache.) In other words, the partial update needs to occur only if the dirty bit is 1 and the partial update's MAC/CTR value is different than that in the cache block. Note that this is also safe to do since we merely use the partial update value to decide to persist or not, however the newest counter value is already incorporated in the integrity tree (root). The main issue of the WTBC scheme is its need for fine-grain dirtiness tracking of security metadata, which can add extra storage overhead to caches.

To avoid adding the area overhead, we propose our second design, which we call **Write-Back Through Status Checks (WTSC)**. WTSC is based on our observation that upon the insertion of an entry to the PUB, we can use the block dirty status to determine if the partial update value is captured during the eviction of an older partial update entry or not. Specifically, if the metadata block is already dirty upon a new partial update, it means that a prior partial update occurred

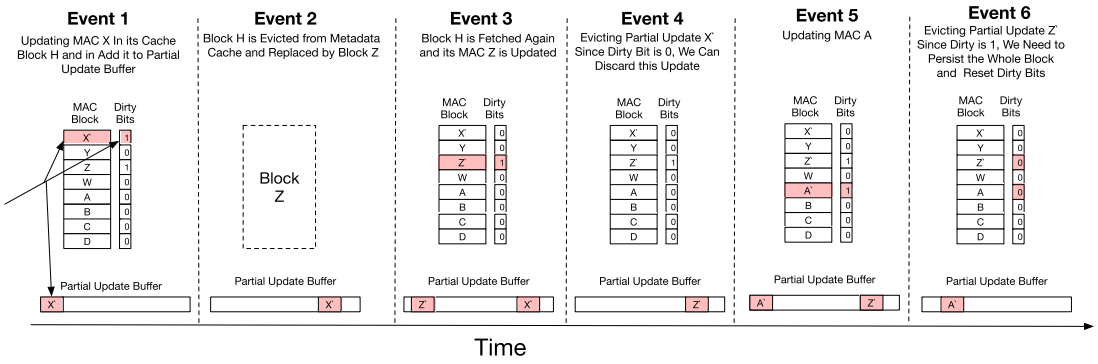

Fig. 5. Demonstration of how WTBC captures the various PUB eviction scenarios. MAC block is shown, but similar events and actions would also occur for counter block.

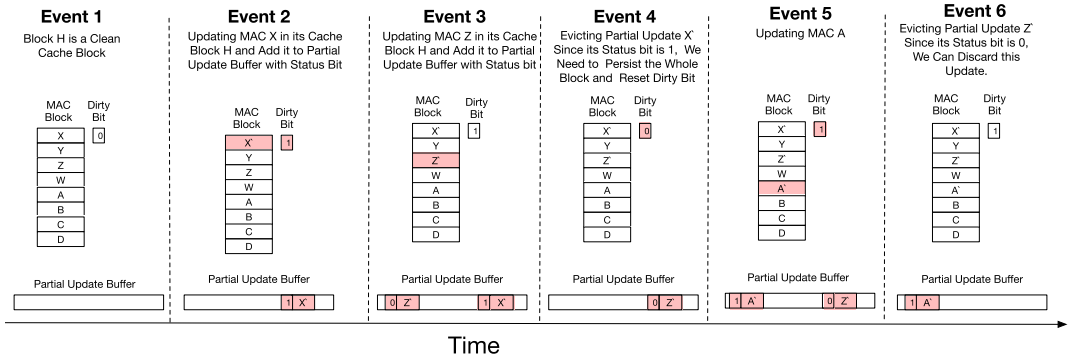

Fig. 6. Demonstration of how WTSC captures the various PUB eviction scenarios. MAC block is shown, but similar event and actions would also occur for counter block.

and has not been evicted yet (otherwise the block would have been cleaned upon its persistence). We observe that upon fetching a metadata block from memory, only the first partial update which converts its status to dirty needs to persist the block upon the partial update's eviction. Meanwhile, for all later partial updates arriving while the block is already dirty, their values will be captured and persisted implicitly upon the persistence of the partial update preceding them which caused the block to become dirty. Thus, we record the dirty bit status upon a partial update along with its entry in the PUB, which we refer to as the *status bit*. However, WTSC partially captures case ① and ②; if a another (or same) metadata partial update within the same data block occurs after eviction of the block, then the status bit will be set to 1 in the corresponding entry. However, upon the eviction of a the later entry, we will needlessly still persist the block even though the new partial update value might have been already captured. Thus, WTSC captures all the cases needed for functional correctness but is more conservative as it fails to precisely detect if the persist is not needed (not the other way around) compared to WTBC. Figure 6 demonstrates how WTSC captures the various scenarios.

Fortunately, we empirically found that WTSC, even though an approximate but needlessly more conservative version of WTBC, is sufficient to eliminate most of the writes upon the eviction of partial updates from the buffer. Thus, for the rest of the paper, we select WTSC as the eviction persist policy of partial updates from the PUB.

C. Interactions with WPQ

Today's processors support a small persistent write buffer on-chip called the write pending queue (WPQ) [21]. The WPQ is typically a small buffer (e.g., 64 entries) and merely holds the evicted blocks before being written to NVM; this reduces the latency it takes to persist data compared to waiting until the block reaches the NVM. Adding the PUB raises the design question of how to interact and leverage the WPQ. First, to persistently combine partial updates, we simply dedicate entries from the ADR-backed WPQ to combine partial updates together. We refer to the WPQ entries used to coalesce independent partial updates as the *persistent combining buffer (PCB)*, and the rest of the WPQ entries as *WPQ*.

Logically, there are two ways to arrange the ordering between PCB and WPQ, PCB-before-WPQ or WPQ-before-PCB. The former waits until the PCB is full before placing the entries in the WPQ using the address of where this block needs to be written in the PUB in memory. However, this approach misses opportunities to coalesce updates to the same security metadata block once it is evicted from the small size PCB; even though partial updates could belong to the same security metadata, they will look like independent entries once inserted in the WPQ (they are tagged with different addresses in WPQ).

The other approach is to place the PCB after the WPQ (PCB-after-WPQ approach). In this approach, WPQ entries are augmented with a volatile (i.e., no need for extra ADR support) bitmask to indicate which partial parts of a metadata block are in this block. Each time there is a partial update, we

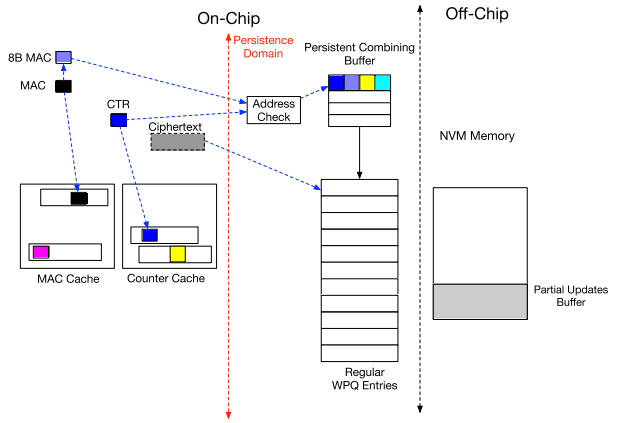

Fig. 7. Overview of PCB-before-WPQ approach

check if the security metadata block exists in the WPQ, and if so, we update the bitmask and merge the partial update within its original metadata block in the WPQ. Upon an eviction of a WPQ entry, we check the bitmask, if all bits are set to 1 (e.g., ciphertext block or metadata block has all its fields updated while in WPQ), then we persist the block in its original location. Otherwise, based on the number of partial updates within a metadata block derived from the bitmask, we can choose to place it in PCB or simply persist the full block.

Fortunately, we found that an augmented version of PCB-before-WPQ, where we check the addresses of partial updates in the PCB upon each partial update such that they are merged, can minimize the pressure on the WPQ and obtain similar performance as in PCB-after-WPQ. Thus, throughout our evaluation we use such an augmented PCB-before-WPQ with 8 entries of the WPQ devoted for PCB while the remaining 56 entries are used as WPQ, compared to a 64-entry WPQ in the baseline. Figure 7 depicts our adopted approach.

D. Recovery Scheme

Upon a crash, the PUB will contain more recent values than the values in the security metadata blocks in their original locations in NVM. Thus, upon a restoration from crash, these security metadata need to be merged with their original locations in NVM (rather than the PUB buffer in NVM). It is critical to note that even though the most recent counters and MACs are not persisted in-place, they can still be verified using the typical mechanism relying on the integrity tree. Thoth relies on prior works to ensure crash consistency of the integrity tree, e.g., Anubis [49], but reduces the number of writes needed to be able to reconstruct the verifiable integrity tree. For instance, in Anubis [49], the counters and MACs must be recovered, however they are verified through a persistent up-to-date integrity root. We leverage the same mechanism, however, before we reconstruct the then-to-be-verified tree, we need to recover the counters and MACs; such recovery of MAC/CTRs was previously done by leveraging ECC bits or co-location, which is no longer feasible in emerging memory interfaces. Thus, **Thoth's responsibility is to merely merge the updates in PUB with the tree (and MAC blocks) to be re-constructed before verification**. To do that, the first step is to scan through the partial updates in PUB in a reverse order (i.e., oldest entry to youngest entry), read it, then read the corresponding metadata blocks, merge the updates (counter and MAC) in their corresponding blocks and write them to memory. To recover the MAC, we will fetch the corresponding ciphertext, compute two levels of MAC, and use the second level of MAC to verify the fetched ciphertext. Note that the potential replay attack will be detected later when the integrity of counters is verified using typical mechanism relying on the integrity tree. Once all PUB entries are incorporated, the tree reconstruction can be done as suggested in Anubis [49]. Note that to speed-up recovery time, Anubis already records the addresses of the blocks lost from the cache in a shadow region. Thus, Thoth would first recover the entries in the WPQ, then leverage Anubis' fast recovery mechanism by reading its shadow address tracking and start the tree reconstruction of the inconsistent parts, all before the tree verification starts.

Thus, in addition to the sub-seconds of recovery time for Anubis [49], we add a marginal extra recovery time of 7 seconds even for a PUB as large as 64MB[5]. We believe that 7 seconds of recovery time is marginal compared to other boot-up and OS aspects upon system startup.

V. Evaluation

A. Simulation Setup

We use GEM5 [9], a cycle-level simulator to evaluate the performance overheads of Thoth. As illustrated in Table I, we simulate 4 X86-64 Out-of-Order cores with 32GB DDR-based PCM. We also use 4 database benchmarks from WHISPER [33] and one in-house benchmark (Random Array Swap). For each benchmark, we fast-forward to a point where the application has run at least 5000 transactions on each core and also insert partial update entries into the PUB using realistic secure metadata generated during the fast-forwarding phase. In the evaluation, we start evicting entries from the PUB when it is 80% full and this threshold is met after the fast-forwarding phase. We set the off-chip PUB size to be 64MB, which incur less than 1% storage overhead with off-chip memory capacity of 32GB. We incorporate a counter cache, MAC cache and MT cache with default size of 64 kB, 128 kB and 256 kB, respectively, in the memory controller. We model a 10-level MT over NVM with lazy update and a 4-level MT over the secure metadata cache with eager update. In our model, a single MAC computation takes 40 cycles, similar to prior work [6], [29], [49]. Table I lists the detailed configuration for our simulation. For all applications, we use 128B as the default transaction size. The WHISPER workloads are command-line configurable for multiple transaction sizes, and we implement our in-house benchmark with similar functionality by setting the swapped array length to the transaction size.

Baseline machine setup: The baseline machine adopts strict persistence for counters and MACs and allows persisting MT nodes through natural evictions, since these can be verified

[5]This can be calculated by latency needed to each of the PUB blocks, reading their corresponding MAC, ciphertext and counter blocks, computing 2 leves of MAC, then updating the counter and MAC blocks of each PUB blocks' entries.

TABLE I
SIMULATION CONFIGURATION PARAMETERS

Processor	
Core	4 Cores, X86, OoO, 4GHz
L1 Cache	2 cycles, 64KB,2-Way
L2 Cache	20 Cycles, 2MB, 8-Way
LLC	32 Cycles, 16MB, 16-Way
WPQ size	64 entries in baseline; 56 entries in Thoth
DDR based PCM Memory	
Size	32 GB
Access Latency	read latency 150ns. write latency 500ns.
Secure Memory Parameters	
AES Latency	40 Cycles
Single Hash Latency	40 Cycles
Integrity Tree	10-level 8-ary Merkle Tree over NVM; 4-level 8-ary Merkle Tree over secure cache
Tree Update Policy	Lazy Update for MT over NVM Eager Update for MT over secure metadata cache
Partial Update Buffer	WTSC, size = 64MB
Persistent Combining Buffer	8 entries
Number of partial updates / cache block	9 updates in 128B block; 19 updates in 256B block;
Counter cache size	64kB; 4 way
MAC cache size	128kB; 8 way
Merkle Tree cache size	256kB; 8 way

using an eagerly updated root [49]. Since each counter is persisted directly to NVM, besides updating the 4-level small merkle tree over the secure metadata cache, we calculate another hash for the last level of the merkle tree. We set the WPQ to start draining when it is 50% full so that secure metadata from the same cache block that arrive in a short time period can be coalesced.

B. Overall Performance

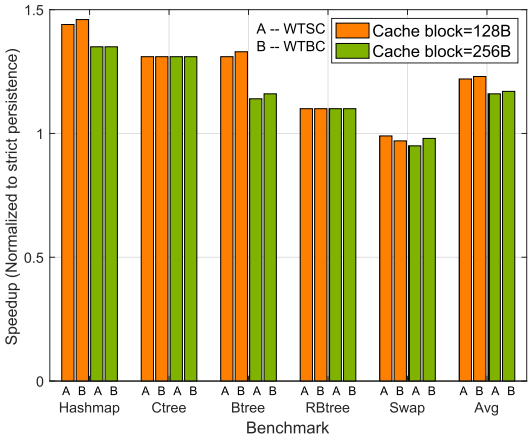

Fig. 8. Speedup of Thoth with WTSC and WTBC scheme (transaction size = 128B)

Figure 8 compares the overall speedup achieved by Thoth over the baseline for both 128B and 256B cache block sizes. Each workload is configured with a transaction size of 128B. Thoth achieves similar speedup using WTSC scheme and WTBC scheme. Thoth achieves an averaged speedup of 1.22x and 1.16x for cache block sizes of 128B and 256B, respectively. The performance advantages of Thoth stem largely from the reduction in write traffic compared to the baseline, as

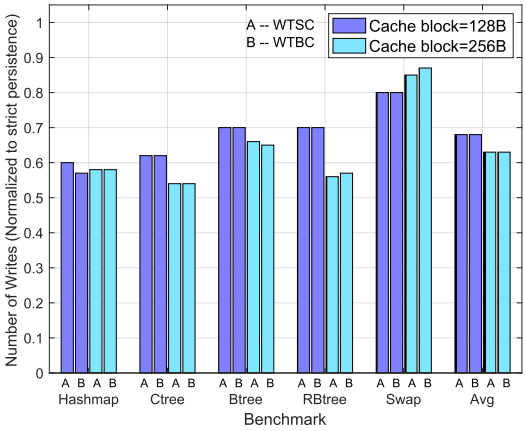

Fig. 9. Number of Writes of Thoth with WTSC and WTBC scheme (128B transactions)

shown in Figure 9. Thoth reduces the number of writes by an average of 32% and 37% for block size of 128B and 256B, respectively, over the baseline system. The 256B blocks are able to pack more partial updates per block and coalesce more entries, leading to a greater reduction in writes. To explain memory reduction in Thoth, we break down write requests in both baseline design and Thoth design. In the baseline design, the write requests can be classified into three main categories: 1. Writes for regular data; 2. Writes for counter blocks; 3. Writes for MAC blocks. In Thoth, the write requests can be classified into four main categories: 1. Writes for regular data; 2. Writes for PCB entries; 3. Evicted counter blocks; 4. Evicted MAC blocks. (There exist other categories. Since their numbers are low, we do not list them here.) In the baseline, categories 2 and 3 take an average of 24.37% and 29.7% respectively from the total write requests. In Thoth, categories of 2, 3, 4 takes an average of 3.95%, 6.81%, 9.46% respectively from the total write requests. The swap benchmark shows the opposite behavior(i.e., more reduction of writes in 128B cache block) because more evictions of secure metadata in Thoth with 256B cache blocks. Even though that higher reduction in writes for 256B cache blocks, the speedups are less for 256B cache block. This is because writes in baseline are reduced by 256B cache block. Therefore, the performance of our baseline is improved by using 256B cache block, which leads to less speedup in Thoth using 256B cache block.

Among the workloads, the swap benchmark does not achieve any speedup when using Thoth and even degrades in performance a little. This is partly an effect of the relatively small transaction size of 128B. Because swap merely exchanges two arrays that are allocated contiguously in memory, it touches few memory locations and induces relatively few secure metadata writes to memory as compared with the other workloads. While Thoth successfully eliminated 20% and 15% of swap's writes to memory, for 128B and 256B respectively, this did not yield a speed-up.

C. Sensitivity to Different Transaction Size

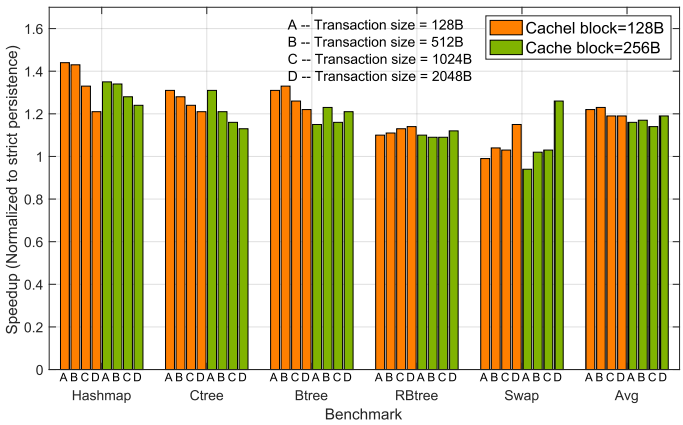

Fig. 10. Speedup of Thoth for transaction sizes of 128B, 512B, 1024B, 2048B

We also study the performance of Thoth over multiple transaction sizes. We run each application with transaction sizes of 128B, 512B, 1024B and 2048B using cache block sizes of 128B and 256B. The average speedup of Thoth with 128B blocks is 1.22x, 1.23x, 1.19x and 1.19x when transaction size is 128B, 512B, 1024B and 2048B respectively, as shown in Figure 10. The average speedup of Thoth with 256B blocks is 1.16x, 1.17x, 1.14x and 1.19x when transaction size is 128B, 512B, 1024B and 2048B respectively, as shown in Figure 10. In some of the benchmarks, Thoth achieves less speed up over the baseline as transaction size increases because the baseline behavior improves. Larger transactions create more opportunity for coalescing secure metadata writes in the WPQ in the baseline machine, reducing the gains in Thoth.

TABLE II
AVERAGED PERCENTAGE OF WRITES FOR CIPHERTEXT ON TRANSACTION SIZE OF 128B, 512B, 1024B, 2048B

Type(Cache block)	Percentage of merged partial updates			
	128B	512B	1024B	2048B
Baseline(Cache block=128B)	45.52%	49.28%	52.68%	58.15%
Baseline(Cache block=256B)	41.35%	44.15%	47.24%	51.30%
Thoth(Cache block=128B)	68.35%	67.97%	68.78%	73.59%
Thoth(Cache block=256B)	67.09%	67.00%	69.60%	76.24%

TABLE III
AVERAGED PERCENTAGE OF PARTIAL UPDATES MERGED IN PCB ON TRANSACTION SIZE OF 128B, 512B, 1024B, 2048B

Cache block	Percentage of merged partial updates			
	128B	512B	1024B	2048B
Cache block = 128B	74.36%	57.68%	44.26%	34.25%
Cache block = 256B	87.88%	80.51%	71.17%	62.74%

To help explain this trend, Table II shows the average percentage of writes to ciphertext for both the baseline and Thoth. The percentage of writes for ciphertext in Thoth mainly depends on the eviction rate from the secure metadata cache and the coalescing effect in the PCB. As the transaction size increases, the relative benefit of coalescing in the PCB decreases with respect to coalescing in the WPQ in the baseline, as shown in Table III. This is because the on-chip PCB can only coalesce consecutive updates for the same partial update (same minor counter/MAC). With larger transaction sizes, these consecutive updates are less likely to reside in the PCB at the same time. However, the WPQ can coalesce any two updates to the same secure metadata cache block, giving it more flexibility to coalesce writes. Also, the WPQ is larger than the PCB, giving it another advantage of more entries over which it can attempt to coalesce. In spite of these advantages in the baseline, Thoth still achieves a significant speed up across all configurations. Thoth reduces the number of writes on an average by 32%, 28%, 24% and 20% for 128B cache block and 37%, 33%, 31% and 31% for 256B cache block. The trend shows Troth reducing writes by a smaller percentage as the transactions grow larger because of better coalescing in the WPQ of the baseline machine as transaction size increases.

Btree of 256B cache block, rbtree and swap have a different trend. They achieve more speedup as transaction size increases due to the number of writes in the baseline and/or more reduction in writes caused by less evictions from metadata cache with larger transaction size. In some cases(e.g., Swap) where performance increases with transaction size, the number of memory writes with larger transaction size in the baseline is significantly higher than the one in smaller transaction size, which signify the impact of memory writes reduction on performance.

D. Sensitivity to Secure Metadata Cache Size

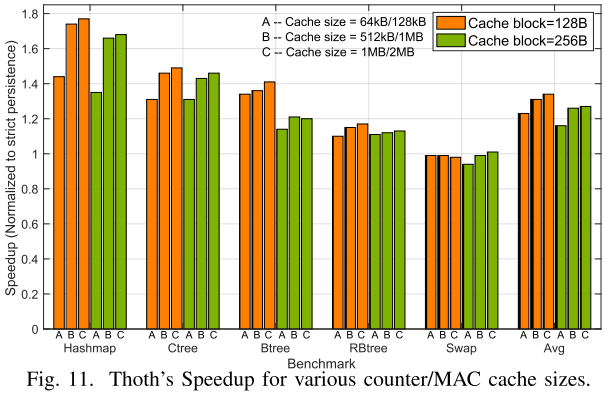

Fig. 11. Thoth's Speedup for various counter/MAC cache sizes.

We further characterize Thoth in the context of larger secure metadata caches. We evaluate the performance of Thoth when varying the counter/MAC cache size to 64kB/128kB, 512kB/1MB and 1MB/2MB. The average speedup of Thoth with a 128B cache block is 1.22x with the smallest secure metadata cache sizes to 1.34x at the largest sizes as shown in Figure 11. Similarly, the average speedup with a 256B cache block is 1.16x for the smallest cache sizes up to 1.28x for the largest. These trends show that Thoth achieves even more speedup with larger metadata cache sizes. This is because Thoth allows secure metadata to be persisted through natural eviction from cache, and with larger metadata cache sizes, there will be fewer evictions and, hence, fewer write backs as well.

E. Sensitivity to Reduced WPQ Size

We also study the performance of Thoth over smaller WPQ with 16 entries and 32 entries. In the experiments, we reserve 1/8 of WPQ entries for the purpose of PCB. We assumed 64 entries WPQ, however when using smaller WPQ with 32 entries and 16 entries, we observe a 1.48x and 1.65x improvement respectively with 128 byte cache block and 1.50x and 1.81x improvement respectively with 256 byte cache block, as shown in Figure 12. Thoth achieves more speedup with smaller WPQ because less security metadata is coalesced in the baseline with smaller WPQ.

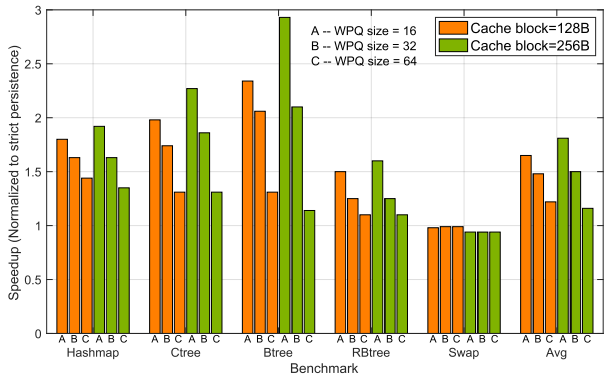

Fig. 12. Thoth's Speedup for various WPQ size.

F. Compare Thoth with Mechanism Using ECC Bits

In this section, we compare Thoth with Anubis [49]. Anubis leverages and re-purposes ECC bits in memory interface to persist secure metadata. Based on our results, with 64kB(counter) and 128kB(MAC) metadata cache size, Thoth brings an overhead of merely 7% on average over baseline that hypothetically assumes all future chips will have co-located ECC (i.e., Anubis).

VI. Related Work

Secure NVM: In addition to enabling crash-consistency, various secure NVM works optimize for different things. For instance, Anubis [49], enables fast and ultra-low recovery time of integrity trees and counters. Meanwhile, Osiris [44] allows recovery of encryption counters regardless of recovery time. Soteria [50] hardens integrity trees to improve reliability. Dolos [19] minimize the latency for persistent transactions through a decoupled security unit. Janus [29] enables efficient scheduling of security memory backend operations. All these works need a new vehicle to persist their security metadata when no ECC or extra metadata bits are available for the processor, and hence Thoth can help realizing them in future memory interfaces.

Overloading ECC Bits: Most of the state-of-the-art solutions in secure NVM [6], [10], [19], [42]–[44], [47], [49]–[51] builds on the idea of co-locating such security metadata with data to minimize the write amplification for implementing crash-consistent secure NVMs. Other works, focus on selectively choosing which data to persist [28]. Thoth provides a new vehicle for prior works through leveraging off-chip partial update buffers in the absence of processor-controlled ECC bits or extra pins, as the case in future memory interfaces. For demonstration purposes, we presented Thoth enabling a state-of-the-art secure NVM work [49] in future interfaces. Some other works [51] leveraged WPQ-like structures to hold metadata blocks before they are persisted, to enable merging more writes before writing to NVM; this is already captured in our baseline.

VII. Conclusion

In this paper, we propose Thoth, a novel off-chip persistent buffer that can enable crash consistency in future memory interfaces with minimal write amplification. Thoth works by combining several partial secure metadata updates into one memory block write and persists them in a large off-chip persistent buffer in NVM. Based on our evaluation, Thoth improves the performance by an average of 1.22x (up to 1.44x) while reducing write traffic by an average of 32% (up to 40%) compared to the baseline Anubis when adapted to future interfaces.

Acknowledgment

Part of this work was funded through Office of Naval Research (ONR) grants N00014-21-1-2809 and N00014-21-1-2811, and the National Science Foundation (NSF) grants CNS-1717486, CNS-1814417, CNS-1908471 and CNS-2008339. The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Approved for public release. Distribution is unlimited.

References

[1] "Enabling Intel Optane DC Persistent Memory on Lenovo ThinkSystem Servers," "https://lenovopress.com/lp1167.pdf", [Online; accessed 07-November-2021].

[2] "Intel hardware sheild – intel total memory encryption," [Online; accessed 09-November-2021]. [Online]. Available: {"https://www.intel.com/content/dam/www/central-libraries/us/en/documents/white-paper-intel-tme.pdf"}

[3] M. Alwadi, K. Zubair, D. Mohaisen, and A. Awad, "Phoenix: Towards ultra-low overhead, recoverable, and persistently secure nvm," IEEE Transactions on Dependable and Secure Computing, pp. 1–1, 2020.

[4] AMD. (2021) Amd memory encryption. [Online; accessed 7-Nov-2021]. [Online]. Available: https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf

[5] Arm. (2021) Arm confidential compute architecture. [Online; accessed 7-Nov-2021]. [Online]. Available: https://www.arm.com/why-arm/architecture/security-features/arm-confidential-compute-architecture

[6] A. Awad, M. Ye, Y. Solihin, L. Njilla, and K. A. Zubair, "Triad-nvm: Persistency for integrity-protected and encrypted non-volatile memories," in 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA), 2019, pp. 104–115.

[7] 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, ser. ASPLOS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 263–276. [Online]. Available: https://doi.org/10.1145/2872362.2872377

[8] L. A. Barroso, J. Clidaras, and U. Hölzle, The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition, 2013. [Online]. Available: http://dx.doi.org/10.2200/S00516ED2V01Y201306CAC024

[9] 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, no. 2, p. 1–7, Aug. 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[10] Z. Chen, Y. Zhang, and N. Xiao, "Cachetree: Reducing integrity verification overhead of secure nonvolatile memories," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 40, no. 7, pp. 1340–1353, 2021.

[11] Chenyu Yan, D. Englender, M. Prvulovic, B. Rogers, and Yan Solihin, "Improving cost, performance, and security of memory encryption and authentication," in 33rd International Symposium on Computer Architecture (ISCA'06), 2006, pp. 179–190.

[12] S. Chhabra and D. Durham, "METHOD AND APPARATUS FOR SHARING SECURITY METADATA MEMORY SPACE," in United States Patent Application 20200183861.

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

[14] T. Dinh Ngoc, B. Bui, S. Bitchebe, A. Tchana, V. Schiavoni, P. Felber, and D. Hagimont, "Everything you should know about intel sgx performance on virtualized systems," Proc. ACM Meas. Anal. Comput. Syst., vol. 3, no. 1, mar 2019. [Online]. Available: https://doi.org/10.1145/3322205.3311076

[15] J. Frazelle, "Power to the people: Reducing datacenter carbon footprints," Queue, vol. 18, no. 2, p. 5–18, apr 2020. [Online]. Available: https://doi.org/10.1145/3400899.3402527

[16] A. Freij, S. Yuan, H. Zhou, and Y. Solihin, "Persist level parallelism: Streamlining integrity tree updates for secure persistent memory," in 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2020, pp. 14–27.

[17] S. Gueron, "A memory encryption engine suitable for general purpose processors," Cryptology ePrint Archive, Report 2016/204, 2016, https://ia.cr/2016/204.

[18] 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," Commun. ACM, vol. 52, no. 5, p. 91–98, May 2009. [Online]. Available: https://doi.org/10.1145/1506409.1506429

[19] X. Han, J. Tuck, and A. Awad, "Dolos: Improving the performance of persistent applications in adr-supported secure memory," in MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1241–1253. [Online]. Available: https://doi.org/10.1145/3466752.3480118

[20] A. Hassan, H. Vandierendonck, and D. S. Nikolopoulos, "Energy-efficient hybrid dram/nvm main memory," in 2015 International Conference on Parallel Architecture and Compilation (PACT), 2015, pp. 492–493.

[21] Intel. (2020) Build persistent memory applications with reliability availability and serviceability. [Online; accessed 7-March-2021]. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/build-pmem-apps-with-ras.html

[22] Intel. (2021) Persistent memory programming. [Online; accessed 7-Nov-2021]. [Online]. Available: https://pmem.io/pmdk/

[23] Intel, "Does Intel® Server System M50CYP Support eADR (Enhanced Asynchronous DRAM Refresh)?" {https://www.intel.com/content/www/us/en/support/articles/000088236/server-products/single-node-servers.html}, 2022, [Online; accessed 7-March-2022].

[24] F. Ishibashi, "Introducing Optanedc persistent memory," [Online; accessed 07-November-2021]. [Online]. Available: {"http://www.ipsj.or.jp/sig/os/index.php?plugin=attach&refer=ComSys2019&openfile=ComSys2019-IntelDCPMMver1.0.pdf"}

[25] 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, vol. 30, no. 1, pp. 143–143, 2010.

[26] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in Proceedings of the Fifteenth European Conference on Computer Systems, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387532

[27] Linux, "Linux Direct Access of Files (DAX)," 2019, [Online; accessed 22-July-2019]. [Online]. Available: "https://www.kernel.org/doc/Documentation/filesystems/dax.txt"

[28] 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), 2018, pp. 310–323.

[29] S. Liu, K. Seemakhupt, G. Pekhimenko, A. Kolli, and S. Khan, "Janus: Optimizing memory and storage support for non-volatile memory systems," in Proceedings of the 46th International Symposium on Computer Architecture, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 143–156. [Online]. Available: https://doi.org/10.1145/3307650.3322206

[30] Micron. (2021) Introducing micron® ddr5 sdram: More than a generational update. [Online; accessed 7-Nov-2021]. [Online]. Available: https://www.micron.com/-/media/client/global/documents/products/white-paper/ddr5_more_than_a_generational_update_wp.pdf?la=en

[31] Microsoft. (2021) Intel optane dc persistent memory, azure netapp files, and azure ultra disk for sap hana. [Online; accessed 7-Nov-2021]. [Online]. Available: https://azure.microsoft.com/en-us/blog/intel-optane-dc-persistent-memory-azure-netapp-files-and-more-for-sap-hana/

[32] P. J. Nair, D.-H. Kim, and M. K. Qureshi, "Archshield: Architectural framework for assisting dram scaling by tolerating high error rates," in Proceedings of the 40th Annual International Symposium on Computer Architecture, ser. ISCA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 72–83. [Online]. Available: https://doi.org/10.1145/2485922.2485929

[33] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," SIGARCH Comput. Archit. News, vol. 45, no. 1, p. 135–148, Apr. 2017. [Online]. Available: https://doi.org/10.1145/3093337.3037730

[34] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," SIGARCH Comput. Archit. News, vol. 37, no. 3, p. 24–33, Jun. 2009. [Online]. Available: https://doi.org/10.1145/1555815.1555760

[35] 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 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007), 2007, pp. 183–196.

[36] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, and M. K. Qureshi, "Synergy: Rethinking secure-memory design for error-correcting memories," in 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2018, pp. 454–465.

[37] S. Van Doren, "Abstract - hoti 2019: Compute express link," in 2019 IEEE Symposium on High-Performance Interconnects (HOTI), 2019, pp. 18–18.

[38] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao, "Characterizing and modeling non-volatile memory systems," in 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2020, pp. 496–508.

[39] M. Ware, K. Rajamani, M. Floyd, B. Brock, J. C. Rubio, F. Rawson, and J. B. Carter, "Architecting for power management: The ibm® power7™ approach," in HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture, 2010, pp. 1–11.

[40] T. E. Wire. (2021) Global non-volatile memory (nvm) market 2021 size, industry share, growth drivers, current trends and future scope, business prospect with regional analysis forecast to 2027. [Online; accessed 7-Nov-2021]. [Online]. Available: https://www.theexpresswire.com/pressrelease/Global-Non-Volatile-Memory-NVM-Market-2021-Size-Industry-Share-Growth-Drivers-Current-Trends-and-Future-Scope-Business-Prospect-with-Regional-Analysis-Forecast-to-2027_14447534

[41] J. Xu and S. Swanson, "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, ser. FAST'16. USA: USENIX Association, 2016, p. 323–338.

[42] F. Yang, Y. Chen, H. Mao, Y. Lu, and J. Shu, "Shieldnvm: An efficient and fast recoverable system for secure non-volatile memory," ACM Trans. Storage, vol. 16, no. 2, May 2020. [Online]. Available: https://doi.org/10.1145/3381835

[43] F. Yang, Y. Lu, Y. Chen, H. Mao, and J. Shu, "No compromises: Secure nvm with crash consistency, write-efficiency and high-performance," in

2019 56th ACM/IEEE Design Automation Conference (DAC), 2019, pp. 1–6.

[44] M. Ye, C. Hughes, and A. Awad, "Osiris: A low-cost mechanism to enable restoration of secure non-volatile memories," in 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2018, pp. 403–415.

[45] S. Yoo, D. Kim, Y. M. Koo, S. K. Wooju Jeong, H. Shim, W.-J. Lee, B. S. Lee, S. Lee, H. Choi, H. D. Lee, T. Kim, and M.-H. Na, "Structural and device considerations for vertical cross point memory with single-stack memory toward cxl memory beyond 1x nm 3dxp," in 2022 IEEE International Memory Workshop (IMW), 2022, pp. 1–4.

[46] V. Young, P. Nair, and M. Qureshi, "Deuce: Write-efficient encryption for non-volatile memories," ACM SIGPLAN Notices, vol. 50, pp. 33–44, 05 2015.

[47] J. Zhou, A. Awad, and J. Wang, "Lelantus: Fine-granularity copy-on-write operations for secure non-volatile memories," in 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), 2020, pp. 597–609.

[48] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in Proceedings of the 36th Annual International Symposium on Computer Architecture, ser. ISCA '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 14–23. [Online]. Available: https://doi.org/10.1145/1555754.1555759

[49] K. A. Zubair and A. Awad, "Anubis: Ultra-low overhead and recovery time for secure non-volatile memories," in 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA), 2019, pp. 157–168.

[50] K. A. Zubair, S. Gurumurthi, V. Sridharan, and A. Awad, "Soteria: Towards resilient integrity-protected and encrypted non-volatile memories," in MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1214–1226. [Online]. Available: https://doi.org/10.1145/3466752.3480066

[51] P. Zuo, Y. Hua, and Y. Xie, "Supermem: Enabling application-transparent secure persistent memory with low overheads," in Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 479–492. [Online]. Available: https://doi.org/10.1145/3352460.3358290