Horus: Persistent Security for Extended Persistence-Domain Memory Systems

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*—Persistent memory presents a great opportunity for crash-consistent computing in large-scale computing systems. The ability to recover data upon power outage or crash events can significantly improve the availability of large-scale systems, while improving the performance of persistent data applications (e.g., database applications). However, persistent memory suffers from high write latency and requires specific programming model (e.g., Intel's PMDK) to guarantee crash consistency, which results in long latency to persist data. To mitigate these problems, recent standards advocate for sufficient back-up power that can flush the whole cache hierarchy to the persistent memory upon detection of an outage, i.e., extending the persistence domain to include the cache hierarchy.**

In the secure NVM with extended persistent domain(EPD), in addition to flushing the cache hierarchy, extra actions need to be taken to protect the flushed cache data. These extra actions of secure operation could cause significant burden on energy costs and battery size. We demonstrate that naive implementations could lead to significantly expanding the required power hold-up budget (e.g., 10.3x more operations than EPD system without secure memory support). The significant overhead is caused by memory accesses of secure metadata. In this paper, we present Horus, a novel EPD-aware secure memory implementation. Horus reduces the overhead during draining period of EPD system by reducing memory accesses of secure metadata. Experiment result shows that Horus reduces the draining time by 5x, compared with the naive baseline design.

***Keywords*-Non-Volatile Memory; eADR; secure memory**

I. Introduction

The ability to retain data in main memory even after power outage or crash events has been an ambitious goal, mainly motivated by the ability to host persistent data (e.g., filesystems) and high-performance crash-recoverable applications. Such applications include key-value store workloads [18], analytical database workloads (i.e., large-scale in-memory analytics) [26], transactional databases [31], graph algorithms [29], etc. ***Persistent Memory (PM)*** generally refers to memory that can retain its content after losing power. Persistent memory can be realized in different ways varying from solutions as expensive as sufficient battery to flush the whole DRAM content to a flash storage, e.g., in NVDIMM-N, to solutions that leverage the high-capacity and persistence features of emerging non-volatile memories (e.g., Intel's DCPMM and JEDEC's NVDIMM-P standard). The latter approach has generally received more interest due to its low access latency, high-capacity exposed to the system, and the reduced costs and area overheads for battery or back-up power source (e.g., large capacitor). Major vendors developed programming frameworks and software development kits for writing persistent applications and workloads which can take advantage of persistent memory [6], [17]. For instance, Intel's PMDK Library [17] allows programmers to write their own persistent applications and algorithms that leverage PM to enable seamless recovery from system crashes. However, persistent applications need to ensure that their updates reach the persistence domain before being considered durable; in its simplest form, the persistence domain is merely the persistent memory module itself. Thus, updates in the processor's volatile caches (or anywhere within the processor chip) could be lost and hence need to be flushed to the persistent memory explicitly by applications.

To guarantee crash consistency, persistent applications explicitly flush durable updates from the volatile caches in the processor chip to the persistent domain followed by a memory fence instruction to force their ordering. The combined usage of flush and fence instructions severely hurts the performance if the persistent domain includes only non-volatile memory, mainly due to its high write latency. Therefore in recent processors, the persistence domain is extended to include a battery-backed Write Pending Queue (WPQ) inside the processor chip. When there is a power outage or crash, WPQ entries are flushed to NVM using the power provided by the battery/capacitor. This feature of extra power to flush WPQ entries is called Asynchronous DRAM Refresh (ADR). A new feature called enhanced Asynchronous DRAM Refresh (eADR), with larger battery than ADR, further extends the persistent domain to include the cache hierarchy [14]. Hereinafter, we refer to systems that include such a feature (including cache hierarchy in persistence domain) by ***Extended Persistence Domain (EPD)*** memory systems. EPD systems aim to improve programmability [1], [25], [30] and enable persistent (i.e., crash recoverable) applications with DRAM-like performance; persistent applications no longer need to flush their persistent updates to the memory controller, but instead it is sufficient to do the update while ensuring cache durability [1].

As NVMs increase the attack window due to their data remanence capabilities, and due to their expected deployment in cloud systems and data centers, it is essential to enable their

usage in trusted execution environments. Fortunately, there have been many recent proposals to enable security primitives (i.e., memory encryption and integrity verification) with emerging NVMs [3], [4], [11], [34], [37], [38]. A large body of prior works [4], [11], [34], [37] focus on careful handling of security metadata for persistent memory; specifically, how to handle security metadata updates efficiently while ensuring security [34], high-availability [37], [38], and crash recoverability [3], [4], [37]. The main challenge of handling security metadata persistently is that much of the updates for security metadata occur in volatile metadata caches, however their corresponding data update is persisted to the memory and hence causes crash inconsistency between data and security metadata upon a crash. Prior works efficiently eliminate large percentage of the performance and write overheads to ensure such consistency in traditional and conventional persistence domain (power-backed write queue through ADR) systems [3], [4], [7], [11], [33], [37].

The Problem: Unfortunately, none of the prior work considers secure memory in systems with EPD (e.g., Intel's eADR support). The cost, area and complexity of the hold-up power support required to back-up the cache hierarchy heavily depends on the maximum number of operations required upon detection of outage. There is an increasing trend to minimize the carbon footprint in data centers [8], which will significantly increase with more operations (hence battery size) needed upon main power source outage. In unprotected systems, i.e., no memory security support, such number of operations will be dominated by those needed writes for every dirty cache line in the cache hierarchy to the memory. In other words, the backup power source should be keeping the system on until all the (dirty) cache hierarchy content is flushed to the persistent memory. Since platform requirements account for the worst case, such budget should be sufficient to flush all the cache hierarchy, i.e., assuming all blocks in caches are dirty. However, for EPD systems with memory security support, flushing the cache hierarchy additionally include two parts: (1) security operations, that could encounter extra accesses, for each memory write (i.e., cache line flushing), and (2) flushing the security metadata cache content securely. Both (1) and (2) heavily depends on the security metadata update scheme. However, we observe that the worst case scenario for completing (1) and (2) can cause **10.3x** more memory operations compared to EPD designed without memory security support.

The Challenge: Ensuring persistent security in EPD systems is challenging due to the following reasons: (1) at *run-time* we aim for a DRAM-like memory security performance, which ignores security metadata persistence operations, e.g., uses lazy update scheme. However, we observe that even using DRAM-optimized secure memory implementations that are meant for non-persistent memories would still incur significant increase in the number of memory accesses required to drain the cache hierarchy, (2) with such implementations, flushing security metadata cache becomes insufficient unless we synchronize its content to update the integrity tree root. Finally, the increase in NVM memory capacity would also increase the number of levels, and hence increases the worst case number of operations required for draining cache hierarchy. Hence, ideally we need a solution that decouples the required backup power budget from the memory capacity, ensures high-performance memory security operations at run-time, incurring minimal extra power requirement compared to no security EPD systems, and ensures fast recovery of the system upon power connectivity.

Accordingly, the goals of this paper are (1) define the backup power budget requirements for encrypted and integrity-verified EPD memory systems, (2) identify the trade-offs for back up power budget, run-time performance overheads, and recovery time (i.e., availability) for enabling crash-consistent and secure EPD systems, and (3) introduce novel mechanisms that reduce the overheads for ensuring recoverability while requiring minimal increase of the backup power budget of non-secure EPD systems. To this end, this is the first work that investigates the impact of combining extended persistence domain and secure memory, which both are likely to be an integral part of future cloud systems and data centers. To define the backup power requirements, unlike traditional studies, we focus on the worst case scenario number of operations (e.g., memory writes) required upon the detection of a crash, and hence the amount of time the processor needs to be up upon a detection of crash. Hence, in our study we focus on optimizing the platform requirements to reduce the costs and area of encrypted and integrity-verified EPD memory systems, however without incurring run-time performance overheads or significant increase in recovery time.

In this paper, we propose a novel scheme, ***Horus***, which reduces the number of write and read requests of secure metadata and avoids updating the regular merkle tree during an EPD flush (i.e. the flush on crash of the EPD state). Horus adopts a split approach for data flushed during crash, different from the approach adopted during run time. It makes the security operation during the crash independent of the regular secure metadata, therefore, it avoids the memory requests of regular secure metadata and avoids updating the regular integrity tree. To evaluate Horus, we use Gem5 simulator [5], an open-source cycle-level simulator, to simulate different scenarios during EPD flush. Our simulation results show that Horus can reduce the memory requests by 8x, reduces the number of MAC calculations by 7.8x, and accordingly reduces the overall system draining time by 5x.

The rest of the paper is organized as follows: Section II provides a relevant background. Section III introduces the motivation. Section IV describes the design details of Horus. Section V shows the evaluation methodology and experiment results. Section VI describes the related work. Section VII concludes the paper.

II. Background

In this section, we discuss the main concepts related to secure persistent memory.

A. Extended Persistence Domain (EPD)

Persistent memory programming frameworks, e.g., Intel's PMDK [17], provides applications with an interface to persist their data. In its most preliminary form, persistent memory is restricted to actual NVM memory module itself, and hence a persist operation would need to wait until the data is flushed all the way to NVM. Unfortunately, this approach suffers from two main limitations, (1) high performance overhead due to the high write latency of NVM, hence slow persist operations, and (2) asymmetry between global visibility (typically starts from cache hierarchy) and persistence domain boundaries, which complicates the programming and concurrency models of persistent applications. To address the high performance overheads, Intel processors include a small buffer, namely Write Pending Queue (WPQ), inside the processor-side memory controller as a part of the persistence domain [13]. This is enabled through extra back-up power, e.g., ultra-capacitor or residual power[1]. Thus, it is sufficient to consider a flush/persist operation completed once the data reaches that buffer.

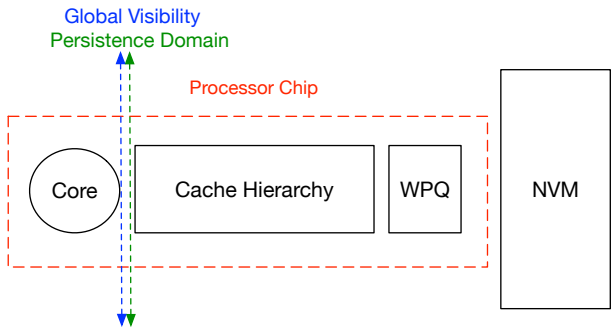

Figure 1. Extended Persistence Domain (EPD).

However, the asymmetry between the global visibility point and the persistence point complicates the programming model and concurrency in persistent applications [1], [14], [25], [30]. Hence, recent NVM systems can leverage a further enhanced persistence domain, we dub as Extended Persistence Domain (EPD), which includes the whole cache hierarchy in the persistence domain. In other words, upon the detection of a power outage, there is enough back-up power to flush the content of cache hierarchy to the persistent memory. As shown in Figure 1, such system merges the global visibility and persistence domains. Intel refers to such feature as Enhanced Asynchronous DRAM Refresh (eADR), and it is expected to be a common feature in new servers. However, the obvious limitation for such EPD systems is the extra power hold-up capabilities (e.g., batteries) required which in turn increase the cost, area, and complexity of the power supply. Existing eADR requires more than 10ms hold-up time Power Supply Unit (PSU), which is expected to increase with gigantic cache hierarchies. For instance, AMD's recent EPYC CPUs are expected to support 512MB L3 cache. Thus, it is critical to incur the minimum number of operations upon crash while preserving the appearance of a ***persistent cache hierarchy***.

[1]This platform feature later became a requirement for persistent memory platform, and is called Asynchronous DRAM Refresh (ADR).

B. Memory Encryption & Integrity Verification

In most trusted execution environments, e.g., Intel's Software Guard Extension (SGX) [15], two critical data protections can be offered: (1) ***Data Confidentiality***, and (2) ***Data Integrity***. Both integrity and confidentiality of data have been emphasized as necessary protection measures for secure execution environments. Hence, many prior studies aim to optimize their performance and write overheads [4], [9], [10], [11], [20], [21], [34], [35], [37], [38]. Below, we briefly describe how these two primitives are commonly implemented.

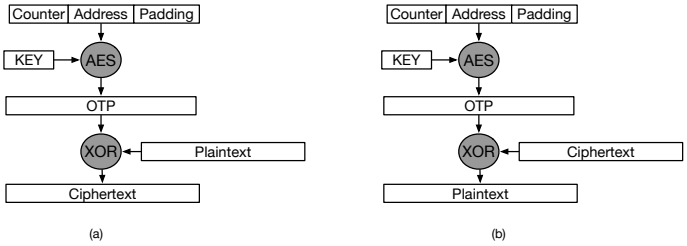

Figure 2. (a) Counter Mode Encryption. (b) Counter Mode Decryption.

Confidentiality: To protect data confidentiality, counter-mode encryption (CME) is generally used due to its security and performance advantages over other schemes [23], [32]. In CME, as shown in Figure 2 ,each data block is associated with a counter that gets incremented each time the block is written to memory. Rather than feeding the plaintext into the encryption engine (e.g., AES), CME encrypts the counter, concatenated with other information (e.g., address), to generate a one-time pad (OTP) which is merely bit-wise XOR'ed with the plaintext/ciphertext to complete the encryption/decryption. In case of a write operation, the corresponding counter will also be encrypted, and hence ensures temporal uniqueness of the encryption pad. Moreover, since the address is used along with counter value to generate the OTP, spatial uniqueness is guaranteed as well, i.e., similar value blocks in different locations appear as different ciphertexts in memory. On the performance side, caching counters in the processor chip can enable overlapping the generation of OTP and fetching the ciphertext from memory; note that only a simple one cycle XOR operation is needed to complete the decryption once the OTP is ready. Since each data block (i.e., cache block size data) has its own counter, the storage overhead can be significant. The

counter should be sized to be large enough such that it never overflows, otherwise the security of CME can be compromised. Meanwhile, its size should be as small as possible to minimize storage overheads.

The state-of-the-art schemes use a split-counter scheme where a major counter (64 bits) is shared among 64 data blocks and each has its own 7-bit minor counter. Thus, each block will use the concatenation of the major and its minor counter to form its own counter. Meanwhile, once a minor counter overflows, the major counter is incremented and all the 64 data blocks sharing the major counter are re-encrypted. By using this scheme, a 64B counter block contains a 64-bit major counter and 64 minor counters, hence covers the counter information for a 4KB region. By caching counter blocks near the memory controller, in what is called counter cache, the spatial and temporal locality of counters can be exploited.

Integrity Verification: One caveat when using CME is the need to protect the integrity of the encryption counters, to avoid counter replay/reuse attacks; CME strictly prohibits the reuse of the same OTP (i.e., per-block counter) for encryption, otherwise it can be easily compromised through known-plaintext attacks [23], [32], [34]. However, since these overall size of these counters relatively large to store on-chip, they are stored off-chip. Hence, it is necessary to verify the integrity of these counters upon fetching them from memory to complete encryption/decryption. Similarly, the data integrity should be protected against attempts to tamper with or replay the content of the secure memory.

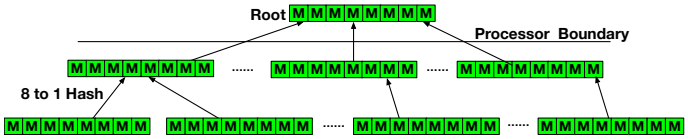

Figure 3. Merkle Tree Structure

To protect the integrity, cryptographic hashes (i.e., message authentication codes (MACs)) are generally used. However, since these MACs need to be maintained per data block to allow fast integrity verification, their total size becomes prohibitive to maintain on-chip. Thus, MACs of data and counters should be maintained off-chip, and hence the MACs also need to be protected. Accordingly, to solve this issue, ***integrity trees (also known as Merkle Tree (MT))*** are generally used. In integrity trees, hashes/MACs are calculated over groups of lower-level MACs to form a tree that ends with a single root value MAC. The root can be thought of as the single hash/MAC value that reflects the content of the whole memory, and is typically stored on-chip (in a persistent register on-chip in case of persistent memory). Upon updating a leaf (e.g., data or counter blocks), their corresponding path and root are also updated. To do verification, each level can be verified through its parent level (in case it is verified), however, if the parent is off-chip, then it also needs to be verified. Fortunately, eventually in worst case scenario where all levels in the path are off-chip and hence need to be verified, the root will be used to verify its children of interest, and then its children will verify its children, etc., until the data/counter block is verified. Figure 3 shows a sample Merkle Tree (MT).

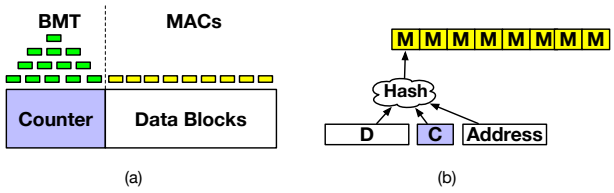

Figure 4. Bonsai Merkle Tree.

Two major optimizations are commonly used to reduce the overheads of MT. The first is based on the observation that protecting the integrity of encryption counters through integrity tree is sufficient to ensure freshness of data when it is accompanied with MACs calculated over the ciphertext, counter and address [23], as shown in Figure 4 (b). In other words, the MT only covers the encryption counters, whereas data blocks have MAC values that can be used to verify their integrity and freshness, as shown in Figure 4 (a). Such a scheme is called Bonsai Merkle Tree (BMT) [23] and is used in nearly all recent works in secure memory [2], [3], [4], [10], [11], [21], [24], [37], [38]. The second optimization is having security metadata cache on-chip that holds the most recently accessed and verified integrity tree nodes (including counter blocks) and data MACs. By doing so, given the high spatial locality of integrity tree nodes, the verification step can be done quickly.

C. Integrity Tree Cache Update Scheme

In the presence of a metadata cache to hold recently accessed integrity tree nodes, MT can be updated by using ***eager*** or ***lazy*** update schemes. ① **Eager update scheme:** Each update on the leaf node triggers updating the whole affected tree path up to and including the root. ② **Lazy update scheme:** A tree node is updated upon eviction of its children from secure metadata cache. When a counter block or an intermediate MT node is evicted, its immediate parent node needs to be updated. In the case of a miss in the secure metadata cache, the immediate parent node needs to be fetched from memory and also the upper level nodes on the affected path until the first hit in the secure metadata cache to verify the integrity of the to-be-updated immediate parent node. In general, lazy update scheme performs better than eager update scheme since lazy update scheme reduces the latency to update all levels of MT. However, the root of MT in lazy update scheme is not always consistent with the leaf nodes. Therefore, in secure NVMs, the root value cannot be used to verify the integrity of leaf nodes after a crash. To allow using lazy update scheme in secure NVMs, an eagerly-updated small MT is applied over secure metadata

cache during run-time. After a crash, we can first recover the secure metadata cache by using mechanisms such as Osiris [34] and Anubis [37]. Later, use the root of the small MT to verify its integrity.

D. Persistent Security

Security metadata state should be consistent with persistent memory content upon a crash, otherwise secure and correct recovery would fail. However, ensuring that the integrity tree, corresponding MAC, and the corresponding encryption counter, all are updated persistently (i.e., in memory) can incur significant overheads. Such overheads stem from the need to push all corresponding security updates to the persistence domain atomically along with the data. When such persistence domain is merely the memory module, or even the module along with small processor-resident buffer (e.g., WPQ), such metadata updates preceding the completion of a persistent transaction can cause significant delays [11]. Similarly, these updates can lead to significant increase in the number of memory writes (and hence premature wear-out). Accordingly, large number of prior work focused on reducing the number of persistent metadata updates to ensure crash consistency [4], [20], [34], enabling fast recovery after a crash [3], [37], reliability of security metadata [38], and ensuring fast integrity tree updates and hence speed up persistent transactions [10], [11].

One common assumption of all prior works is the lack of or a basic ADR support, hence a small buffer in the memory controller is included in the persistence domain. Thus, much of the efforts have been towards ensuring minimum work before inserting data in the persistence domain. However, in EPD systems, the insertion of data in the persistence domain is directly in the critical path as it corresponds to every write operation in the cache hierarchy (including L1 cache updates); the whole cache hierarchy is assumed persistent. Hence, the persistent security support is shifted towards ensuring that flushing the content of the cache hierarchy is done in a crash-consistent way with its corresponding security metadata. Oblivious to the amount of back-up power needed upon outage, one can safely assume that the cache hierarchy contents can be protected (i.e., encrypted and integrity-verifiable) and all security metadata updates can be flushed. However, as we will show in the next section, the amount of extra work needed to enable the aforementioned solution can be prohibitive, especially with most data center vendors aiming to minimize their carbon footprints and the maintenance/deployment challenges for per-server batteries [8].

III. Motivation

In this section, we demonstrate the challenges for implementing EPD systems with secure memory. As mentioned earlier, memory encryption and integrity verification are critical primitives in trusted execution environments. Such protections aim to protect the content of the memory during run-time and across crashes. In non-secure EPD memory systems, the cache hierarchy is flushed line by line to the memory, then the system is considered persisted and ready to go off. In case of inclusive last-level cache (LLC), the process can be simplified to flush all the dirty lines in LLC, since the coherence protocol will bring in any more recent version from upper-level caches. Hence, the EPD power hold-up budget should be designed to be sufficient to drain the maximum number of cache lines that can be dirty in the cache hierarchy, which can be as large as the number of cache lines in the LLC. Note that this can be significant when gigantic caches are used, e.g., AMD EPYC's 512MB V-Cache.

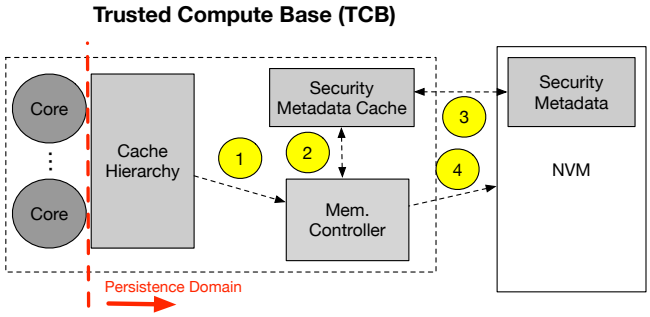

Figure 5. Draining the contents of cache hierarchy in secure EPD systems.

Similarly, in secure EPD memory systems, the contents of the cache hierarchy need to be flushed upon the detection of a power outage. However, each cache line needs to have its integrity (including freshness) and confidentiality protected all the time, including after power recovery. In its simplest form, draining the contents of a secure EPD memory system can be comprised of two steps: (1) draining the contents of the cache hierarchy while performing encryption, MAC calculation and integrity tree update on each cache line flush; i.e., treat LLC flushes similar to run-time main memory writes. Later, (2) flush/synchronize the security metadata cache updates, e.g., integrity tree and counter caches, to the persistent memory. Note that step (2) heavily depends on whether a lazy update or eager update schemes are used. For simplicity, let's assume that step (2) takes negligible number of operations compared to step (1). As shown in Figure 5, draining the cache hierarchy lines consists of the following steps for each eviction. (1) A cache line flushed from the cache hierarchy arrives to the memory controller. (2) Some time later, the memory controller needs to encrypt the line and update the integrity tree (whether lazily or eagerly) to ensure freshness protection, also updating the encryption counter and the MAC written with data regardless of whether such updates are done in the security metadata cache or also persisting to memory. Note that to complete this step, there could be memory accesses required to fetch the security metadata (e.g., counter blocks, BMT path, and/or MAC block) to complete

the protection. Even worse, fetching these metadata blocks to the metadata cache can lead to evicting other (possibly dirty) blocks from the metadata cache and cause additional memory writes. We refer to this as Step (3) in the figure. Finally, once the metadata needed to complete the protection of the cache line are fetched and updated (whether in the cache or persistently), the cache line can be written to memory (step (4)).

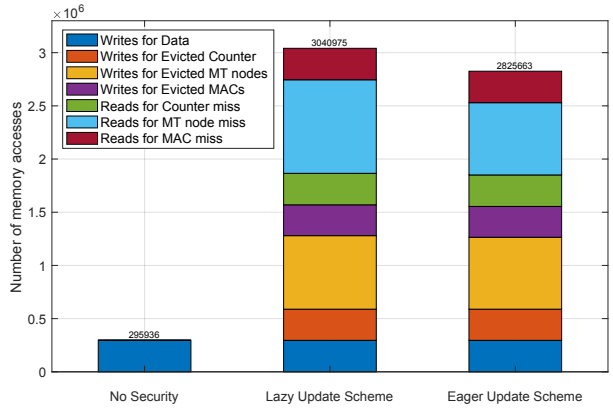

Figure 6. Breakdown of memory requests for flushing cache hierarchy in different scenarios (Total flushed cache blocks = 295936)

To better understand the performance implications of providing security during flushing cache hierarchy, Figure 6[2] compares the number of memory accesses incurred by flushing the cache hierarchy in a system without security, with two systems with security (eager update scheme or lazy update scheme for Merkle Tree). Note that even for merely flushing the cache content securely, secure EPD memory systems need 10.3x and 9.5x more memory accesses (hence more power hold-up budget) when lazy and eager integrity tree update schemes are used, respectively. We assume the cache hierarchy content before crash was randomly filled with sparse contents hence poor spatial locality in security metadata cache. Note that since EPD systems need to be designed with the worst case in mind, e.g., all cache lines are dirty in LLC for the case of non-secure EPD, we also need to consider the worst case for secure EPD and hence accounting for different access patterns. While we do not claim that randomly sparse cache hierarchy contents that are all dirty is even the worst case yet, we have observed an explosion in the required EPD power hold-up budget that is sufficient to motivate the need for novel solutions to enable secure EPD memory systems with reasonable backup power requirements.

IV. Horus Design

In this section, we first describe our threat model. Later, we describe a baseline support for secure EPD memory systems. After that, we discuss two different designs of Horus to guarantee memory security when there is a crash, but with minimal increase in EPD backup power requirements.

A. Threat Model

Our threat model is similar to state-of-the-art works in secure NVM [3], [4], [10], [11], [21], [34], [37], [38] . Specifically, the boundary of our trusted block is the processor chip. We assume that attackers can mount external attacks on the off-chip memory, such as bus-snooping attacks, physical theft, memory scanning, memory replay and memory spoofing. Side-channel attacks such as power-side channel attacks, speculative loads attacks, memory timing attacks, and access pattern leakage are beyond the scope of this work.

B. Baseline Secure EPD Systems

Since this is the first work to explore secure NVM in EPD systems, we start with defining a baseline implementation.

In secure but non-EPD systems (i.e., extremely limited or no backup power at all), the security metadata is updated in a persistent and recoverable way upon each write to persistent memory. To do so, schemes such as eager update (e.g., used in Triad-NVM [4]) always update the root of the tree before completing the data write. Hence, once the integrity tree is rebuilt, the system is considered recovered and can be verified using the up-to-date root on-chip. However, such a scheme can lead to high recovery time due to rebuilding the whole tree, and hence works such as Anubis [37] book-keeps which parts need to be rebuilt. On the other hand, while eager update is simpler to implement, lazy update scheme is generally faster as it only updates the leaf nodes on each write. Only upon the eviction of dirty nodes (including leaf nodes) do the updates propagate to the parent which will be placed in the cache and marked dirty. However, the lazy update scheme leads to crash consistency issues since the root will be stale upon system recovery. Prior works have presented mechanisms to enable recovery with lazy update scheme, however at the cost of extra complexity and memory writes [3], [37].

In non-persistent main memory systems (e.g., DRAM-based), either lazy or eager update schemes can be used without the need for any extra work to ensure recoverability. Secure EPD memory systems present an interesting design point where we aim for recovery-oblivious performance at run-time (due to sufficient power budget) but also need to flush sufficient amount of information upon detection of a crash to allow secure and consistent recovery.

Secure EPD Memory Systems: One possible design point is to use recovery-oblivious secure memory implementation, similar to those used with non-persistent memory. For instance, merely a lazy update or eager update schemes without any extra steps to ensure per write crash consistency. As shown in Figure 7, during run-time the EPD system can run with secure memory mode similar to non-persistent

[2]See the methodology section for more information about the assumed contents of the cache hierarchy upon crash. Note that EPD platform requirements must account for the worst case.

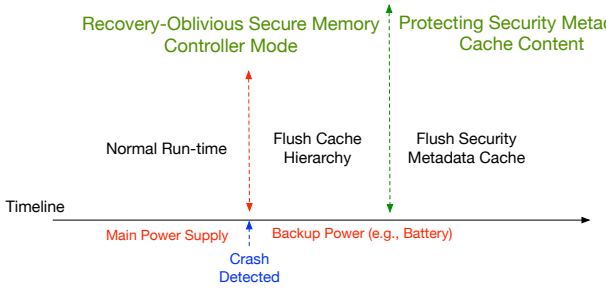

Figure 7. Timeline of events for in baseline secure EPD system.

memory systems, i.e., recovery-oblivious, and hence minimal performance overheads. However, once a crash is detected, the EPD draining mode is triggered and an alternative power supply will be engaged (e.g., battery or capacitor). The first step is to write the contents of the cache hierarchy to memory while still using the recovery-oblivious secure memory mode. In other words, cache lines evicted from cache hierarchy will be encrypted and integrity-verifiable, however their most-recent security metadata might be not persisted yet (hence crash inconsistency). Thus, to solve the issue, once the cache hierarchy is flushed, the security metadata cache contents need to be either synchronized and flushed to their locations in memory or simply protected (e.g., using a small integrity tree) and flushed to a reserved region in memory. Leveraging EPD power, if eager update was used during run-time, then just flushing the security metadata cache to their original places will be sufficient. However, if lazy update scheme was used during run-time, then one way would be to scan through all metadata cache contents, bring their ancestor path (after verification) and propagate the update all the way through the root. However, this is costly, and hence an alternative approach is to calculate a single hash value over the metadata cache content, using a small integrity tree, similar to Anubis [37]. Later, flush the metadata cache content to a reserved region in memory. Upon recovery, the metadata content will be restored from memory and verified.

Unfortunately, we observe that the step of flushing cache hierarchy can significantly increase the EPD power hold-up budget; as shown in Section III, it requires more than 10.3x and 9.5x additional memory accesses when recovery-oblivious lazy and eager update schemes, respectively, are used. Note that if the cache hierarchy is needlessly flushed with recovery-awareness (e.g., Anubis [37]), then we would incur even more memory accesses to flush the cache hierarchy.

C. Horus

As discussed earlier, flushing the cache hierarchy while operating the secure memory controller in run-time mode can lead to significant increase in the number of memory accesses and operations. We also notice that such massive increase in the number of memory accesses results from security metadata fetches and updates, especially when the cache hierarchy before the crash was filled with dirty sparse cache lines, hence maximize the number of misses in the security metadata cache. Thus, our main ***design objective*** is to minimize the maximum number of extra operations needed upon flushing the cache hierarchy. Note that the power supply requirements are defined based on the worst case number of operations needed upon outage detection.

The first insight we leverage in Horus design is that in-place flushing of cache hierarchy contents is what leads to these extra accesses. Specifically, since flushed cache lines are written to their memory locations, then we need to use their address-specific metadata (e.g., BMT nodes, counters and MACs). However, sparse contents of cache hierarchy would naturally lead to many misses in such metadata. On the other hand, in-place updates without updating or using the respective (and verified) metadata would lead to security vulnerabilities (e.g., reuse of counter value) and/or functional errors due to the inability to discriminate which memory content was protected with the main BMT and their corresponding counters. Accordingly, we replace in-place updates when flushing the cache hierarchy with flushing the content to a small reserved region in memory which can be protected by using separate security metadata (i.e., CHV Security Metadata). We refer to this region by ***cache hierarchy vault (CHV)***, as shown in Figure 8. Note that in the baseline secure EPD system we might need to fetch security metadata from memory (in case of miss) since the counters to be used for encryption must be verified for integrity (as shown in Figure 8 part B). On the other hand, CHV's security metadata (shown in Figure 8 part C) is only written during the flushing stage and only brought back upon recovery to verify the contents. Later, in the next section, we will discuss why such counters used for encrypting CHV do not need to be integrity verified.

Our goal is to ensure that the flushed cache hierarchy content to CHV is ① protected in terms of confidentiality and integrity, ② such a protection is implemented using much less number of operations compared to in-place evictions/flushes, and ③ the flushed contents of the cache hierarchy are recoverable (and verifiable as guaranteed by ①). The rest of this subsection describes how each one of these goals is met through our Horus design.

1) Protecting Confidentiality and Integrity of CHV: The CHV should exactly contain all dirty cache blocks in the cache hierarchy at the time of the crash. Such a protection needs to guarantee the following: ① all dirty contents of the cache hierarchy are flushed to CHV. This implies the need for a mechanism to ensure that the number of blocks flushed upon crash detection is exactly the same as those existing in CHV. Otherwise, attackers can selectively omit new memory updates (which was present in cache) and hence replay a previous content. ② ensure that the addresses of the flushed cache blocks are also protected from tampering (including

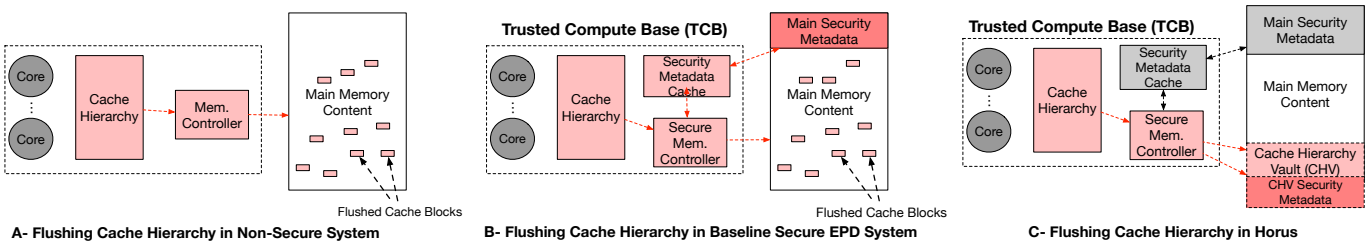

Figure 8. Comparison between the cache draining of (A) non-secure EPD system (B) Baseline secure EPD system, and (C) Horus.

splicing attacks to swap the addresses of different cache blocks flushed to CHV). Note that the confidentiality of the addresses themselves is not protected since these in anyway will be observed later in the memory bus; access pattern leakage are beyond our threat model. However, their integrity protection is a must. ③ the confidentiality and integrity of the blocks written to memory. The cache blocks should be encrypted in a way that hides any temporal or spatial similarity between values as guaranteed during run-time. The temporal leakage could happen through observing the same ciphertext written for the same address or across different addresses over time (including cache draining episodes). In other words, we need to make sure that flushing the exact same content before encryption of the same address or other addresses even in different draining events will lead to different ciphertext written to memory. Finally, the integrity protection should ensure the cache blocks, as well as their addresses, are integrity protected and cannot be replayed from previous CHV contents of a previous draining process. In other words, we need to ensure the *freshness* of the CHV contents.

To this end, the CHV area includes drained **cache blocks** area, **addresses** of the drained cache blocks area, and **security metadata** area that stores the metadata used to protect the integrity and confidentiality of CHV. To provide such protections, one straightforward way is to treat the CHV areas to be protected as a miniature of the main memory, and hence have encryption counters dedicated for each memory block in the CHV area, and protect such counters using a small integrity tree rather than the larger integrity tree used for the main memory. Moreover, the protected CHV memory blocks have MACs co-located with that are calculated over the positional address in CHV, the encrypted memory block, and its corresponding CHV address encryption counter. In other words, a BMT style is used merely for protecting the address and data blocks in CHV. Unfortunately, this scheme requires fetching the CHV-address counters and verify them through the CHV integrity tree before using them, otherwise counter reuse with the same address could occur. Moreover, another limitation is the small BMT fetch and update process on each block flushing during the draining process.

Fortunately, we do not need to persist the CHV counters and a tree to protect them. Specifically, we observe that maintaining a monotonically increasing counter is sufficient to ensure unique initialization vectors for each flush operation upon draining. In particular, a persistent counter, always kept inside the processor chip, is incremented after each flush operation to memory. We refer to such a counter by ***drain counter (DC)***. By additionally book-keeping the latest number of drained blocks from the cache hierarchy persistently, we can know what is the most recent counter used for each flushed block. We refer to that register by ***ephemeral drain counter (eDC)***. The eDC value is cleared after each recovery of the system. By guaranteeing that each flush uses a unique initialization vector, we close both temporal and spatial leakage channels within a single draining process and across multiple draining episodes. Also, since the starting address of the CHV is fixed, we can relate each block flushed in the CHV to its drain counter value (address - CHV Base Address + DC - eDC).

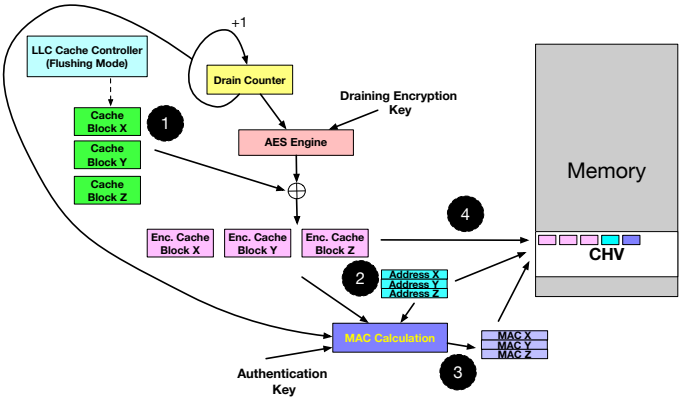

Figure 9. Simplified cache draining steps in Horus.

Figure 9 depicts the steps taken by Horus upon draining the cache hierarchy. As shown in Step ①, the LLC cache controller (or firmware loaded upon power outage) will start flushing the cache contents. Meanwhile, Horus encrypts these blocks using counter-mode encryption where drain counter used as initialization vector. In Step ②, the address of each block flushed is inserted to form a complete 64B block that just contains addresses in order, i.e., 8 addresses (assuming 64-bit addresses). In Step ③, MACs are calculated for each encrypted block along with its address (also stored in the address block) and the drain counter value used to encrypt

it. Similar to addresses, MACs are combined into a single 64B and written to memory as one block. Finally, as shown in Step ④, the encrypted data blocks, address block and MAC block are written in order to the CHV. Note that in the figure we only show three data blocks being flushed, however these steps will repeat until all cache blocks are drained. For simplicity, we omitted the CHV address calculation part from the figure, however it can be simply calculated using the starting address of the CHV and drain counter as discussed earlier.

2) Write-Friendly and Optimized Protection of CHV: To further optimize the cache draining process, we coalesce the MACs calculated for multiple cache blocks (and address blocks) into one MAC block before writing it to the CHV security metadata region. Moreover, since the addresses of cache blocks are smaller than the memory write granularity, we coalesce multiple addresses into one block before writing it to memory. Moreover, to reduce the number of MACs need to be written for CHV integrity protection, instead of book-keeping a MAC value per memory block, we can do that for a coarser granularity. For instance, by merely maintaining two 64B MAC registers, we can use the first register to buffer 8 MAC values which once is full is used to calculate one 8B MAC value that gets buffered in the second register, before the first register is emptied. Once the second register is full, i.e., has eight MAC values, it will be written to memory as a single 64B block before being emptied. In other words, even though not to the extent of a full Merkle Tree, we hierarchically but efficiently (only using two registers) build two levels of MACs but only store the highest level to reduce the number of writes, as shown in Figure 10.

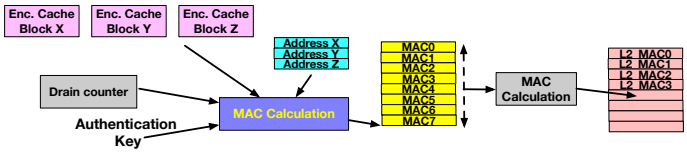

Figure 10. The Horus Double-Level MAC scheme for Horus

Our insight is that storing a MAC per memory block is mainly used to optimize the run-time memory access by avoiding over-fetching, since we do not need to bring neighbouring blocks to complete update/verification. However, in Horus, since it occurs only at draining time, we always have the neighbouring cache contents to be written to CHV. Note that neighbouring cache contents could be spatially far in terms of original memory location, however in CHV they are written contiguously. We dub this optimized scheme as ***Horus Double-Level MAC (Horus-DLM)***. On the contrary, we refer to the default scheme where a MAC is stored per memory block with ***Horus Single-Level MAC (Horus-SLM)***.

3) Recovery of Flushed Cache Content: The recovery process is straightforward. Upon power recovery, the contents in CHV are read back by the processor, in a reversed flush order, to recover the cache hierarchy. Data blocks and their corresponding address and MAC blocks, are read back together. The drain counter value used to encrypt each data block can be derived from its position in CHV along with the most recent value (after the crash) of the persistent *drain counter* on-chip. Upon the decryption and integrity verification of every data block drained to the CHV, we can either place them back in the LLC in dirty state or write them back to their original locations and update the main integrity tree accordingly (i.e., treat them as normal run-time writes). For simplicity, we opt for the first option that reads them back to the LLC and marks each as dirty. We assume the LLC cache controller will be aware that the system is in recovery mode and hence treats any fill operations as dirty blocks. However, if the cache is non-inclusive then to reduce complexity the second option (i.e., fetch, verify then write to memory using the main security metadata) may be used. Note that commercial EPD systems, e.g, eADR, already support flushing all caches in non-inclusive LLC systems, hence the only difference in Horus will be the recovery step in case we decide to place the flushed data blocks upon crash back to the cache hierarchy.

4) Security Analysis: The confidentiality of data blocks flushed to the CHV are guaranteed by ensuring a never repeating drain counter value, and hence unique one-time pads for counter mode encryption. Meanwhile, the integrity of the data blocks and their corresponding address blocks in the CHV can be verified using their MACs also stored in the CHV. Specifically, since the MAC of each data block is calculated using its address, the drain counter value used to encrypt it, and the encrypted data block itself, we can reproduce the MAC and compare it with the one inside the CHV. Thus, if the address block has been tampered with then the MAC generated using the tampered address will mismatch with the stored MAC, and thus will be detected. Similarly, if a data block has been tampered with, then the MAC will also mismatch and hence will be detected. Finally, if the attackers attempt to replay a previous CHV content or splice/swap current contents of the CHV, then the MAC will mismatch because the draining counter value used to protect data written to that location in the CHV will be different than the drain counter value used elsewhere.

D. Hardware Cost of Horus

In this section, we describes the hardware required by Horus. Horus leverages the existing secure memory support(AES, MAC) engines and secure metadata caches which are used during run-time, instead, Horus uses them during draining time. In addition, Horus needs some extra registers: two registers for DC and eDC (with simple ALU logic to increment the counter), one register for coalescing addresses, one register for coalescing MACs(two registers in the case of Horus Double-Level MAC shceme). In addition, Horus

requires reserving a small region of NVM to be used as CHV. The area of CHV is nearly proportional to the cache size, $Size_{CHV} = 1.25 \times Size_{cache} + 1.125 \times Size_{metadata\_cache}$ in the case of Horus-SLM.

V. Evaluation

A. Simulation Setup

Table I
Simulation Configuration Parameters

Processor	
Core	Single Core, X86, OoO, 4GHz
L1 Cache	2 cycles, 64KB,2-Way
L2 Cache	20 Cycles, 2MB, 8-Way
Inclusive LLC	32 Cycles, 16MB, 16-Way
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	160 Cycles
Integrity Tree	a 10-level 8-ary Merkle Tree over NVM; a 5-level 8-ary Merkle Tree over secure cache
Counter cache size	256kB; 8-Way
MAC cache size	512kB; 8-Way
Merkle Tree cache size	256kB; 8-Way

To evaluate the performance of Horus, we use a cycle-level simulator, GEM5 [5]. As illustrated on Table I, we simulate a single X86 core with 32GB DDR-based PCM. In Horus, the draining time is independent of the spatial relationships between the blocks in the cache hierarchy upon crash; however for the baseline, as also shown in the Section III, it heavily depends on the spatial relationship between evictions as this highly impacts the behavior of the integrity tree cache. Since EPD battery requirements depend on the worst case, we assume a cache hierarchy content with extremely poor spatial adjacency between blocks. Specifically, we fill the cache hierarchy with cache blocks that are at least 16KB distant in their physical addresses; the 16KB was derived by dividing the simulated memory size by total size of cache hierarchy. While we do not claim that this is even the worst case, given the idiosyncratic behavior of the lazy update scheme with certain cache configurations, it is sufficient to demonstrate the high battery demands for a naive implementation of secure EPD systems. Meanwhile, Horus does not use integrity tree to drain the cache hierarchy and, hence, is oblivious to its upon-crash contents' spatial characteristics.

For evaluation, we compare the following four schemes: a baseline lazy update scheme (**Base-LU**), baseline eager update scheme (**Base-EU**), Horus with single-level MAC (**Horus-SLM**), and Horus with double-level MAC (**Horus-DLM**).

B. Overall Performance

The major components of the computing system, e.g., processor chip and memory modules, must be powered on until the system securely drains its cache hierarchy contents. Thus, a conservative proxy of the battery requirements in EPD systems is how long the system needs to stay powered on upon crash detection, hence using the alternative power source. Accordingly, we simulate the execution time starting from the detection of possible outage until the whole cache hierarchy is drain (including security metadata cache). Figure 11 shows the difference in execution time to drain the system.

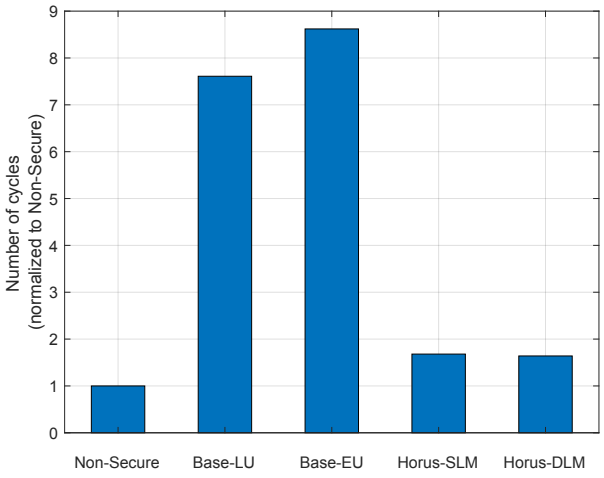

Figure 11. Normalized number of cycles

As shown in the figure, both baselines using eager and lazy update schemes take 5.1x and 4.5x, respectively, longer time to drain the system compared to both Horus schemes. Accordingly, we expect Horus schemes to reduce the battery requirements for secure EPD systems by orders of roughly 4x-5x. Note that the ability to enable commercial EPD support, even without secure memory, is heavily restricted by the power supply power hold-up time capabilities. For instance, Intel's eADR cannot be enabled on systems with less than 10ms power hold-up time [16]. Thus, reducing the hold-up time requirement for secure EPD systems by such significant amount will enable wider adoption of secure memory in future EPD systems. As also shown in the figure, Horus schemes reduces the draining time from 8.6x more time, hence 8.6x higher power hold-up time, to merely 1.7x compared to non-secure EPD system. As expected, the Base-EU takes the longest time to drain due to the large number of MAC operations and memory accesses (to fetch and update a whole integrity tree path on each write request).

C. Write Operations

To better understand the draining time reduction using Horus, Figure 12 shows the total number of memory write operations required for each scheme with a breakdown of the type of write request needed. We can observe that the majority of memory writes in the baseline are due to evictions of integrity tree metadata blocks due to security operations upon draining cache hierarchy. Both Base-EU and Base-LU use

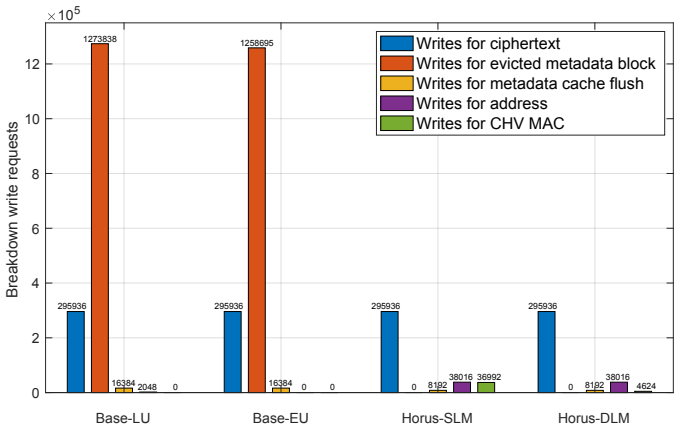

Figure 12. Breakdown of memory writes in different designs.

the main integrity tree upon draining cache, and hence poor spatial locality cache hierarchy contents can lead to massive number of misses in the BMT cache. We can also observe that, as expected, Horus-SLM incurs 8x smaller number of CHV MACs due to its coarser granularity protection. Also, a key observation from the figure is that flushing the metadata cache contents, once the cache hierarchy is drained, is negligible in all schemes. The reason is that the amount of metadata cache dirty evictions when draining the cache hierarchy is more dominant (in case of Base-LU and Base-EU), however even without such extra writes the number of cache blocks in LLC is much larger than metadata blocks. Thus, even in Horus, we can see that the number of metadata block flushes is negligible.

D. MAC Calculations

Another source of overhead during the system draining stage is the MAC calculations needed for authenticating the flushed contents. Figure 13 shows the breakdown of the number of MAC calculations needed for each scheme.

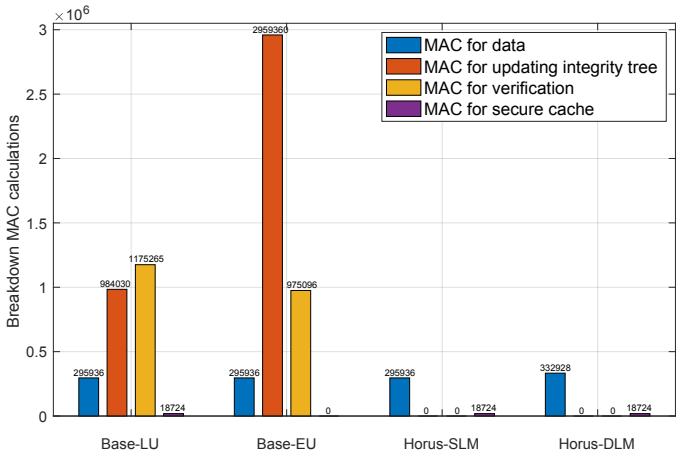

Figure 13. Breakdown of MAC calculations in different designs

As shown in the figure, the eager update baseline (Base-EU) consumes the largest number of MAC calculations. The largest portion of MAC calculations in Base-EU is for updating the integrity tree (the second bar). On the other hand, since the integrity tree is eagerly updated, there is no need for any MAC calculations to protect the integrity tree (the fourth bar), but just merely flushing its content; the tree root is already up-to-date. On the other hand, for Base-LU, the largest contributor for MAC calculations is those used for verification, e.g., MAC to verify the counters and integrity tree nodes. Meanwhile, we can see that for both Horus schemes the most dominant MAC calculations are for the MACs to protect the flushed data blocks from cache hierarchy. Finally, we can observe that Horus-DLM consumes a 1.125x more MACs than that of Horus-SLM, mainly because of calculating a second level MAC to reduce the number of MAC writes.

E. Sensitivity to Cache Size

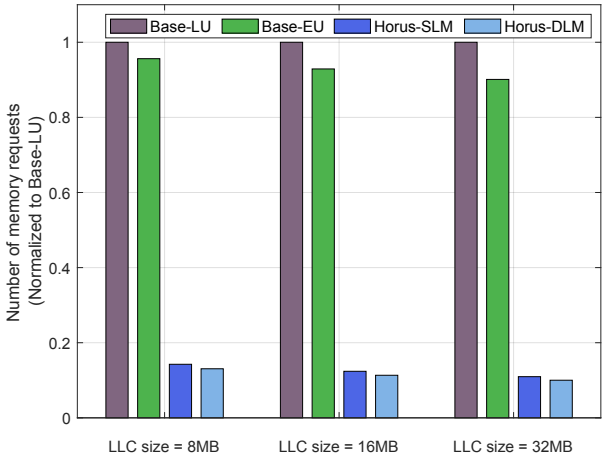

Figure 14. Normalized memory requests required in different Last-Level-Cache sizes of 8MB, 16MB and 32MB

In this section, we show the performance overhead of Horus on variable Last-Level-Cache sizes of 8MB, 16MB and 32MB. Figure 14 and Figure 15 shows the number of memory requests and MAC calculations normalized to Base-LU with corresponding LLC size. With LLC size of 8MB, 16MB and 32MB, both Horus schemes achieve at least 7.0x and 5.8x reduction in memory requests and MAC calculations, respectively, compared to Base-LU design.

F. Estimation of Recovery Time

In this section, we calculate the recovery time of Horus-SLM and Horus-DLM with varied LLC size from 8MB to 128MB. In Horus, the dominant parts of the recovery process are reading back the CHV content, integrity verification and data decryption. Figure 16 shows the recovery time of Horus-SLM and Horus-DLM. The parameters we use to estimate the recovery time are from Table I. We can observe that even

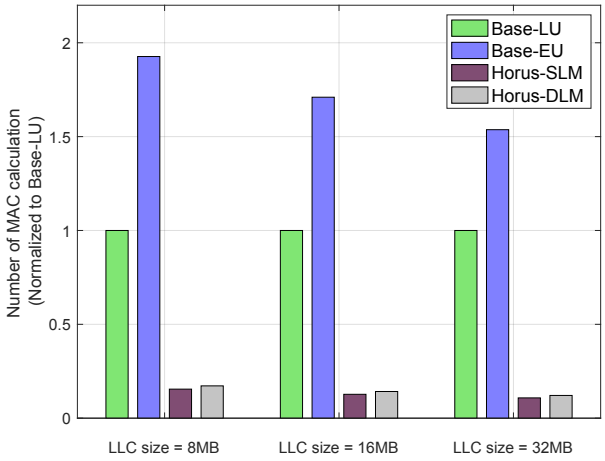

Figure 15. Normalized number of MAC calculations required in in different Last-Level-Cache sizes of 8MB, 16MB and 32MB

for LLC caches as large as 128MB, the recovery time of Horus-SLM and Horus-DLM is extremely small (0.51s and 0.48s, respectively). Hence, we believe that Horus can be used even in systems with very high availability requirements.

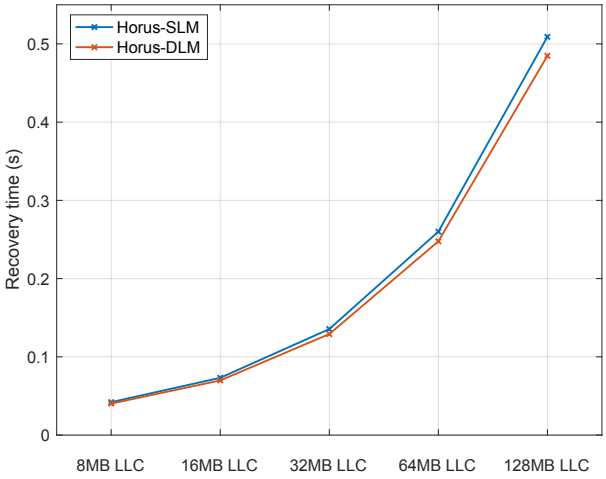

Figure 16. Recovery time of Horus-SLM and Horus-DLM.

G. Estimation of Energy Costs and Battery Size

In this section, we evaluate the impact of Horus on energy cost during draining and the needed battery size. In eADR systems, once a power outage is detected a SMI interrupt will be signaled and a special code will be executed in the processor to flush the caches. Therefore, the processor needs to stay powered on during the whole cache hierarchy draining process. In our energy model, energy cost during draining mainly comes from 4 aspects: processor energy, NVM write operations, NVM read operations and secure operations. We use McPAT [19] to model the processor energy required during draining. We assume single NVM write and read operation takes the energy of 531.8nJ and 5.5nJ respectively [12]. Energy cost for secure operations is minimal, compared with the other three aspects, therefore it is not included in our estimation. Table II categorizes the energy cost of the four designs. As shown in the table, energy cost during draining is dominated by processor energy, which is largely affected by draining time. Energy cost of Base-LU and Base-EU is 4.5x and 5.1x, respectively, higher than both Horus schemes. Such required energy is closely similar to the draining time, because energy costs during draining is dominated by processor energy, which is mostly affected by draining time.

We use the energy cost in Table II to estimate needed battery size. Different battery technologies have different energy densities. In this paper, we estimate two energy sources: super capacitors(SuperCap) and lithium thin-film batteries(Li-thin), using similar approaches as those used in BBB [1]. The energy density for SuperCap and Li-thin is $10^{-4}Whcm^{-3}$ and $10^{-2}Whcm^{-3}$ respectively. Table III shows the battery size needed for the four designs using SuperCap and Li-thin. For both of the battery technologies, Horus reduces the battery size by at least 4.4x, compared with baseline design.

Table II
ESTIMATION OF ENERGY COSTS OF DIFFERENT OPERATIONS DURING DRAINING

	Base-LU	Base-EU	Horus-SLM	Horus-DLM
Processor Energy(J)	10.21	11.54	2.25	2.20
NVM write operations(J)	0.84	0.83	0.2	0.18
NVM read operations(J)	0.008	0.007	0	0
Total(J)	11.07	12.39	2.45	2.38

Table III
ESTIMATION OF BATTERY SIZE NEEDED FOR DRAINING

	Base-LU	Base-EU	Horus-SLM	Horus-DLM
SuperCap (cm^3)	30.7	34.4	6.8	6.6
Li-thin (cm^3)	0.31	0.34	0.07	0.07

VI. RELATED WORK

In this section, we will discuss the related prior work on NVM system with battery-backed on-chip components and non-volatile caches (NVCaches). We will also discuss prior work on how secure NVM system adapts to system with on-chip persistent domain.

NVM system with battery-backed on-chip resources: On-chip persistent domain is achieved by providing battery-backed on-chip components to flush the data to NVM during crash. For example, with battery-backed WPQ in memory controller (ADR solution [13]), persistent domain is extended to WPQ. With the entire cache hierarchy backed with battery(eADR solution [14]), persistent domain is extended to the cache hierarchy. BBB [1] proposed a battery-backed

buffer attached with the L1 cache to hold the persistent data. Data in the battery-backed buffer is flushed to NVM when there is a crash. BBB extends the on-chip persistent domain to the same point as the eADR solution with smaller battery size. Unfortunately, none of the prior works addressed how secure memory can be implemented in such systems.

NVM system with NVCaches: Instead of using battery-backed cache, some work proposes using non-volatile cache (NVCache) [22], [28], [36]. However, NVM technologies that are suitable for cache usage, e.g., Spin-Torque Transfer RAM (STT-RAM), have limited retention time that can vary from seconds to hours depending on other area/performance trade-offs [27]. Thus, the majority of studies use it as regular cache without any persistence guarantees. Horus on the other hand aims to enable extending the persistence domain leveraging minimal back-up power.

Secure NVM system with on-chip persistent domain: Some works [10], [11] have researched how secure NVM system adapts to system with on-chip persistent domain. Dolos [11] assumes a secure NVM system with battery-backed WPQ. It proposes a Minor-Security-Unit to protect the WPQ to allow immediately flushing the WPQ content when there is a crash and also avoid causing large overhead on the performance of persistent application. Bonsai Merkle Forests [10] propose an on-chip non-volatile secure metadata cache for the integrity tree nodes. The single integrity tree is divided into small integrity trees by storing the roots of the small integrity trees in the non-volatile secure metadata cache to speedup updating the integrity tree. None of these prior works explore secure NVMs with on-chip persistent domain including battery-backed cache hierarchy. Unlike the prior works, Horus explores and provides solutions for how to protect the cache hierarchy when the battery-backed cache hierarchy is flushed to NVM during crash. Moreover, this is the first work that identifies the significant increase in the maximum amount of operations that need to be guaranteed power to securely flush the cache hierarchy.

VII. Conclusion

In conclusion, this is the first paper to observe that significant increase in number of memory operations, and hence draining time, for secure EPD systems. To mitigate the impact of such increase on the complexity and capabilities of power supplies required in future computing systems, we presented novel mechanisms to enable secure memory in EPD systems with reasonable increase in power hold-up time. Specifically, proposed Horus, which could effectively reduce the number of memory operations by at least 8x, and number of MAC calculations by 7.8x, compared to a baseline secure memory (using lazy update scheme). With these reductions, the draining time, and hence power hold-up time requirements, is reduced by 5x compared to the baseline secure memory scheme. Accordingly, Horus significantly reduces the required increase in power hold-up time for secure EPD systems.

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] M. Alshboul, P. Ramrakhyani, W. Wang, J. Tuck, and Y. Solihin, "Bbb: Simplifying persistent programming using battery-backed buffers," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 111–124.

[2] M. Alwadi, A. Mohaisen, and A. Awad, "Promt: Optimizing integrity tree updates for write-intensive pages in secure nvms," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 479–490. [Online]. Available: https://doi.org/10.1145/3447818.3460377

[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] 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.

[5] 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

[6] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages Applications*, ser. OOPSLA '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 433–452. [Online]. Available: https://doi.org/10.1145/2660193.2660224

[7] Z. Chen, Y. Zhang, and N. Xiao, "ExtraCC: Improving Performance of Secure NVM with Extra Counters and ECC," in *The 36th International Conference on Massive Storage Systems and Technology*, 2020.

[8] 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

[9] 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.

[10] A. Freij, H. Zhou, and Y. Solihin, "Bonsai merkle forests: Efficiently achieving crash consistency in secure persistent 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. 1227–1240. [Online]. Available: https://doi.org/10.1145/3466752.3480067

[11] 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

[12] M. Hoseinzadeh, M. Arjomand, and H. Sarbazi-Azad, "Reducing access latency of mlc pcms through line striping," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 277–288.

[13] Intel. (2020) Build persistent memory applications with reliability availability and serviceability. [Online; accessed 7-March-2021].

[14] Intel, "eADR: New Opportunities for Persistent Memory Applications," https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html, 2021, [Online; accessed 7-March-2022].

[15] Intel, "Intel Software Guard Extensions (Intel SGX)," https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html, 2021, [Online; accessed 7-March-2022].

[16] 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].

[17] Intel. (2022) Persistent memory development kit. https://pmem.io/pmdk/. [Online; accessed 7-March-2022].

[18] H. T. Kassa, J. Akers, M. Ghosh, Z. Cao, V. Gogte, and R. Dreslinski, "Improving performance of flash based Key-Value stores using storage class memory as a volatile memory extension," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 821–837. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/kassa

[19] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009, pp. 469–480.

[20] 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.

[21] S. Liu, K. Seemakhupt, G. Pekhimenko, A. Kolli, and S. Khan, "Janus: Optimizing memory and storage support for non-volatile memory systems," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 143–156.

[22] S. Mittal, J. S. Vetter, and D. Li, "Lastingnvcache: A technique for improving the lifetime of non-volatile caches," in *2014 IEEE Computer Society Annual Symposium on VLSI*, 2014, pp. 534–540.

[23] 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.

[24] 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.

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

[26] A. Shanbhag, N. Tatbul, D. Cohen, and S. Madden, "Large-scale in-memory analytics on intel® optane™ dc persistent memory," in *Proceedings of the 16th International Workshop on Data Management on New Hardware*, ser. DaMoN '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3399666.3399933

[27] C. W. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. R. Stan, "Relaxing non-volatility for fast and energy-efficient stt-ram caches," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 2011, pp. 50–61.

[28] M. Soltani, M. Ebrahimi, and Z. Navabi, "Prolonging lifetime of non-volatile last level caches with cluster mapping," in *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*, 2016, pp. 329–334.

[29] A. Uyar, S. Akkas, J. Li, and J. Fox, "Intel optane dcpmm and serverless computing," 2021.

[30] W. Wang, *Architectural Support for Persistent Memory Programming*. Persistent Programming In Real Life (PIRL), 2020., 2020.

[31] 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)*. IEEE, 2020, pp. 496–508.

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

[33] 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.

[34] 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.

[35] 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.

[36] 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 *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 421–432.

[37] 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.

[38] 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