Osiris: A Low-Cost Mechanism to Enable Restoration of Secure Non-Volatile Memories

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

Clayton Hughes
Sandia National Laboratories
Albuquerque, USA
chughes@sandia.gov

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

***Abstract*—With Non-Volatile Memories (NVMs) beginning to enter the mainstream computing market, it is time to consider how to secure NVM-equipped computing systems. Recent Meltdown and Spectre attacks are evidence that security must be intrinsic to computing systems and not added as an afterthought. Processor vendors are taking the first steps and are beginning to build security primitives into commodity processors. One security primitive that is associated with the use of emerging NVMs is memory encryption. Memory encryption, while necessary, is very challenging when used with NVMs because it exacerbates the write endurance problem.**

Secure architectures use cryptographic metadata that must be persisted and restored to allow secure recovery of data in the event of power-loss. Specifically, encryption counters must be persistent to enable secure and functional recovery of an interrupted system. However, the cost of ensuring and maintaining persistence for these counters can be significant. In this paper, we propose a novel scheme to maintain encryption counters without the need for frequent updates. Our new memory controller design, *Osiris*, repurposes memory Error-Correction Codes (ECCs) to enable fast restoration and recovery of encryption counters. To evaluate our design, we use Gem5 to run eight memory-intensive workloads selected from SPEC2006 and U.S. Department of Energy (DoE) proxy applications. Compared to a write-through counter-cache scheme, on average, Osiris can reduce 48.7% of the memory writes (increase lifetime by 1.95x), and reduce the performance overhead from 51.5% (for write-through) to only 5.8%. Furthermore, without the need for backup battery or extra power-supply hold-up time, Osiris performs better than a battery-backed write-back (5.8% vs. 6.6% overhead) and has less write-traffic (2.6% vs. 5.9% overhead).

***Index Terms*—Secure Architecture, Non-volatile Memory, ECC, Computer Architecture**

I. Introduction

Emerging Non-Volatile Memories (NVMs) are a promising advancement in memory and storage systems. For the first time in the modern computing era, memory and storage systems have the opportunity to converge into a single system. With latencies close to those of DRAM and the ability to retain data during power loss events, NVMs represent the perfect building block for novel architectures that have files stored in media that can be accessed similar to the way we access the memory system, i.e., through conventional load/store operations. Furthermore, given the near-zero idle power of emerging NVMs, they can bring significant power savings by eliminating the need for the frequent refresh operations needed for DRAM. NVMs also promise large capacities and much better scalability compared to DRAM [1], [2], which means more options for running workloads with large memory footprints.

While NVMs are certainly promising, they still face some challenges that can limit their wide adoption. One major challenge is the limited number of writes that NVMs can endure; most promising technologies, e.g., Phase-Change Memory (PCM), can only endure tens of millions of writes for each cell [3], [4]. They also face security challenges — NVMs can facilitate data remanence attacks [4]–[6]. To address such vulnerabilities, processor vendors, e.g., Intel and AMD, have started to support memory encryption. Unfortunately, memory encryption exacerbates the write endurance problem due to its avalanche effect [4], [6]. Thus, it is very important to reduce the number of writes in the presence of encryption. In fact, we observe that a significant number of writes can occur due to persisting encryption metadata. For security and performance reasons, counter-mode encryption has been used for state-of-the-art memory encryption schemes [4], [6]–[9]. For counter-mode encryption, encryption counters/IVs are needed and they are typically organized and grouped to fit in cache blocks, e.g., Split-Counter scheme (64 counters in 64B block) [10] and SGX (8 counters in 64B block) [7]. One major reason behind that is, for cache fill and eviction operations, most DDR memory systems process read/write requests on a 64B block granularity, e.g., DDR4 8n-prefetch mode. Moreover, packing multiple counters in a single cache line can exploit spatial locality to increase counter cache hit rate.

Most prior research work has assumed that systems have sufficient residual or backup power to flush encryption metadata, i.e., encryption counters, in the event of power loss [4], [9]. Although feasible in theory, in practice, reasonably long-lasting Uninterruptible Power-Supplies (UPS) are expensive and occupy large areas. Admittedly, the Asynchronous DRAM Refresh (ADR) feature has been a requirement for many NVM form factors, e.g., NVDIMM [11], but major processor vendors only limit persistent domains in processors to tens of entries in the *write pending queue (WPQ)* in the memory controller [12]. The main reason behind this is the high cost for guaranteeing a long-lasting sustainable power supply in case of power loss coupled with the significant power needed for writing to some NVM memories. However, since encryption metadata can be on the order of kilobytes or even megabytes, the system-level ADR feature would most likely fail to guarantee its persistence as a whole. In many real-life scenarios, affording sufficient battery backup or expensive power-supplies is infeasible, either due to area, environmental or cost limitations. Therefore, battery-free

solutions are always in demand.

Persisting encryption counters is not only critical for system restoration but is also a security requirement. Reusing encryption counters, i.e., those not persisted during a crash, can result in meaningless data after decryption. Even worse, losing the most-recent counters invites a variety of attacks that rely on reusing encryption pads (counter-mode encryption security strictly depends on the uniqueness of encryption counters used for each encryption). Recent work [13] has proposed atomic persistence of encryption counters, in addition to exploiting application-level persistence requirements to relax atomicity. Unfortunately, exposing application semantics (persistent data) to hardware requires application modifications. Moreover, if applications have a large percentage of persistency-required data, persisting encryption metadata will incur a significant number of extra writes to NVM. Finally, the selective counter persistence scheme can cause encryption pad reuse for non-persistent memory locations, if a state-of-the-art split counter is used, which enables known-plaintext attackers to observe the lost/new encryption pads that will be reused when the stale counters are incremented after crash recovery. Note that such reuse can happen for different users or applications.

In this paper, we propose a lightweight solution, motivated by a discussion of the problem of persisting NVM encryption counters, and discuss the different design options for persisting encryption counters. In contrast with prior work, our solution does not require any software modifications and can be orthogonally augmented with prior work [13]. Furthermore, our solution chooses to provide consistency guarantees for all memory blocks rather than limit these guarantees to a subset of memory blocks, e.g., persistent data structures. To this end, our solution, ***Osiris***, repurposes Error-Correction Code (ECC) bits of the data to provide a sanity-check for the encryption counter used to perform the decryption. By doing so, Osiris can reason about the correct encryption counter even in the event that the most-recent value is lost. Thus it can relax the strict atomic persistence requirement while providing *secure* and *fast* recovery of lost encryption counters. In this paper, we discuss our solution and evaluate its impact on write-endurance and performance. To evaluate our design, we use Gem5 to run a selection of eight memory-intensive workloads pulled from SPEC2006 and U.S. Department of Energy (DoE) proxy applications. Compared to a write-through counter-cache scheme, on average, Osiris can reduce 48.7% of the memory writes (increase lifetime by 1.95x), and reduce the performance overhead from 51.5% (for write-through) to only 5.8%. Furthermore, without the need for backup battery or extra power-supply hold-up time, Osiris performs better than a battery-backed Write-Back (5.8% vs. 6.6% overhead) and has less write-traffic (2.6% vs. 5.9% overhead). In summary, the major contributions of our paper are the following:

- We propose *Osiris*, a novel scheme that provides crash consistency for encryption counters similar to strict counter persistence schemes but with a significant reduction in performance overhead and number of NVM writes, without the need for an external/internal backup battery. Its optimized version, *Osiris-Plus*, further reduces the number of writes by eliminating premature counter evictions from the counter cache.
- We discuss several design options for Osiris that provide trade-offs between hardware complexity and performance.
- We discuss how our scheme can work with and be integrated with state-of-the-art data and counter integrity verification schemes.

The rest of the paper is organized as follows. In Section II, we briefly discuss the main concepts and background relevant to our paper. In Section III, we discuss our own solution, its design options, trade-offs and security impact. Section IV covers our evaluation methods and analysis for Osiris. Later, in Section V, we discuss prior and related work. Finally, we conclude our work in Section VI.

II. Background and Motivation

In this section, we discuss the main concepts that are relevant to our proposed solution, followed by motivation for our work.

A. Background

In this part of the paper, we will discuss emerging NVMs and state-of-the-art memory encryption implementations.

1) Emerging NVMs: Emerging NVM technologies, such as Phase-Change Memory (PCM) and Memristor, are promising candidates to be the main building blocks of future memory systems. Vendors are already commercializing these technologies due to their many benefits. NVMs' read latencies are comparable to DRAM while promising high densities and potential for scaling better than DRAM. Furthermore, they enable persistent applications. On the other hand, emerging NVMs have limited, slow and power consuming writes. NVMs also have limited write endurance. For example, PCM's write endurance is between 10-100 million writes [3]. Moreover, emerging NVMs suffer from a serious security vulnerability: they keep their content even when the system is powered off. Accordingly, NVM devices are often paired with memory encryption.

2) Memory Encryption and Data Integrity Verification: There are different encryption modes that can be used to encrypt the main memory. The first one is the *direct encryption* (a.k.a ECB mode), where an encryption algorithm, such as AES or DES, is used to encrypt each cache block when it is written back to memory and decrypt it when it enters the processor chip again. The main drawback of direct encryption is system performance degradation due to adding the encryption latency to the memory access latency (the encryption algorithm takes the memory data as its input). The second mode, which is commonly used in secure processors, is *counter mode* encryption. In counter-mode encryption, the encryption algorithm (AES or DES) uses an *initialization vector* (IV) as its input to generate a one-time pad (OTP) as depicted in Figure 1. Once the data arrives, a simple bitwise XOR with the pad is needed to complete the decryption. Thus, the decryption latency is overlapped with the memory access latency. In state-of-the-art designs [4], [10], each IV consists of a unique ID of a page (to distinguish between swap space and main memory space), page offset (to guarantee different blocks in a page will get different IVs), a per-block *minor* counter (to make the same value encrypted differently when written again to

the same address), and a per-page *major* counter (to guarantee uniqueness of IV when minor counters overflow).

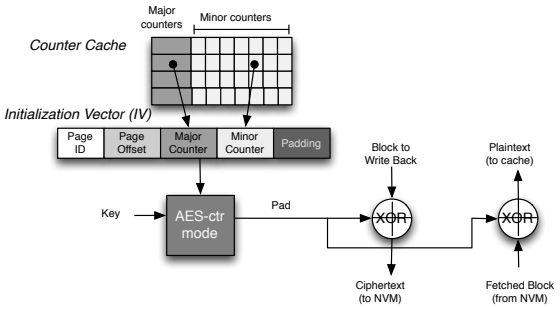

Fig. 1: State-of-the-art memory encryption using counter mode [4].

Similar to prior work [4], [6], [10], [13], [14], we assume counter mode processor-side encryption. In addition to hiding the encryption latency when used for memory encryption, it also provides strong security defenses against a wide range of attacks. Specifically, counter-mode encryption prevents snooping attacks, dictionary-based attacks, known-plaintext attacks and replay attacks. Typically, the encryption counters are organized as major counters, shared between cache blocks of the same page, and minor counters that are specific for each cache block [10]. This organization of counters can fit 64 cache blocks' counters in a 64B block, 7-bit minor counters and a 64-bit major counter. The major counter is only incremented when one of its relevant minor counters overflows, in which the minor counters will be reset and the whole page will be re-encrypted using the new major counter for building the IV of each cache block of the page [10]. When the major counter of a page overflows (64-bit counter), the key must be changed and the whole memory will be re-encrypted with the new key. This scheme provides a significant reduction of the re-encryption rate and minimizes the storage overhead of encryption counters when compared to other schemes such as a monolithic counter scheme or using independent counters for each cache block. Additionally, a split-counter scheme allows for better exploitation of spatial locality of encryption counters, achieving a higher counter cache hit rate. Similar to state-of-the-art work [4], [8]–[10], we use a split-counter scheme to organize the encryption counters.

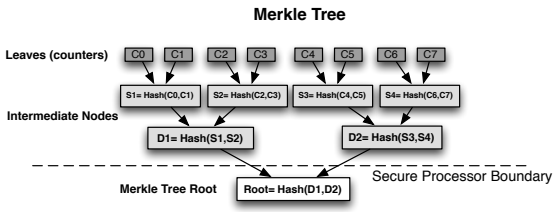

Fig. 2: An example Merkle Tree for integrity verification.

Data integrity is typically verified through a Merkle Tree — a tree of hash values with the root maintained in a secure region. However, encryption counter integrity must also be maintained. As such, state-of-the-art designs combine both data integrity and encryption counter integrity using a single Merkle Tree (Bonsai Merkle Tree [14]). As shown in Figure 2, Bonsai Merkle tree is built around the encryption counters. Data blocks are protected by a MAC value that is calculated over the counter and the data itself. Note that only the root of the tree needs to be kept in the secure region, other parts of the Merkle Tree are cached on-chip to improve performance. For the rest of the paper, we assume a Bonsai Merkle Tree.

B. Encryption Metadata Crash Consistency

While crash consistency of encryption metadata has been overlooked in most memory encryption work, it becomes essential in persistent memory systems. If a crash happens, the system is expected to recover and restore its encrypted memory data. Figure 3 depicts the steps needed to ensure crash consistency.

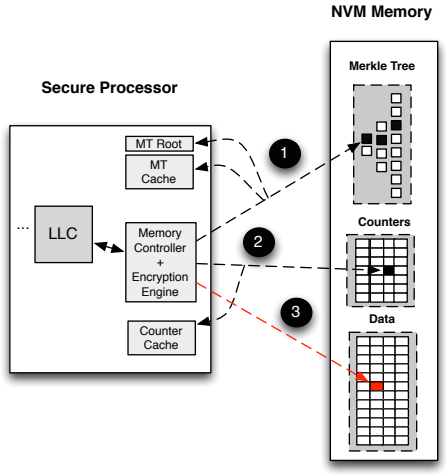

Fig. 3: Steps for write operations to ensure crash consistency.

As shown in the Figure 3, when there is a write operation to NVM, first we need to update the root of the Merkle tree (as shown in step (1)) and any cached intermediate nodes inside the processor. Note that only the *root* of the Merkle Tree needs to be kept in the secure region. In fact, maintaining intermediate nodes of the Merkle Tree can speed up the integrity verification. Persisting updates of intermediate nodes into memory is optional as it is feasible to reconstruct them from leaf nodes (counters) and then generate the root and verify it through comparison with that kept inside the processor. We stress that the root of the Merkle Tree must persist safely across system failures, e.g., through internal processor NVM registers. Persisting updates to intermediate nodes of the Merkle Tree after each access might speed up recovery time by reducing the time of rebuilding the whole Merkle Tree after crash. However, the overheads of such a scheme and the infrequency of crashes make rebuilding the tree a more reasonable option.

In step (2) of Figure 3, the updated counter block will be written back to memory as it gets updated in the counter cache. Unlike Merkle Tree intermediate nodes, counters are critical to keep and persist, otherwise the security of the counter-mode encryption is compromised. Moreover, losing the counter values can result in the inability to restore encrypted memory data. As noted by Liu et al. [13], it is possible

to just persist the counters of persistent data structures (or a subset of them) to enable consistent recovery. However, this is not sufficient from a security point of view; losing counters' values, even for non-persistent memory locations, can cause reuse of an encryption counter with the same key, which can compromise the security of the counter-mode encryption. Furthermore, legacy applications may rely on OS-level or periodic application-oblivious checkpointing, making it challenging to expose their persistent ranges to the memory controller. Accordingly, a secure scheme that persists counter updates and does not require software alteration is needed. Note that even for non-persistent applications, counters must be persisted on each update or the encryption key must be changed and all of the memory must be re-encrypted with a new key. Moreover, if the persistent region in memory is large, which is likely in future NVM-based systems, most memory writes will be accompanied by an operation to persist the corresponding encryption counters, making step ② a common event.

Finally, the written block will be sent to the NVM as shown in step ③. Some portions of step ① and step ② are crucial for correct and secure restoration of secure NVMs. Also note that when updating the root of the Merkle Tree on the chip, updating the counter and writing the data are assumed to happen atomically, either using three internal NVM registers to save them before trying to update them persistently or using hold-up power that is sufficient to complete three writes. To avoid incurring the high latency to update NVM registers for each memory write, a hybrid approach can be used where three volatile registers can be backed with hold-up power to write them to the slower NVM registers inside processor. Ensuring write atomicity is beyond the scope of this paper; our focus is to avoid frequent persistence of updates to counter values in memory and using the fast volatile counter cache while ensuring safe and secure recoverability.

C. Motivation

As mentioned earlier, counter blocks must be persisted to allow safe and secure recovery of the encrypted memory. Also, the Merkle Tree root cannot be written to NVM and must be saved in the secure processor. The intermediate nodes of the tree can be reconstructed after recovery from a crash, hence there is no need to persist updates to the affected nodes after each write operation. In other words, no extra work should be done to the tree beyond what occurs in conventional secure processors without persistent memory. As the performance overhead of Bonsai Merkle Tree integrity verification is negligible (less than 2% [14]), our focus in this paper is to reduce the overhead of persisting encryption counters. To persist the encryption counters, we employ two methods: *Write-Through Counter Cache (WT)* and *Battery-Backed Write-Back Counter Cache (WB)*. In the WT approach, whenever there is a write operation issued to NVM, the corresponding counter block is updated in the counter cache and persisted in NVM before the data blocks are encrypted and updated in NVM to complete the write operation. However, the WT approach causes significant write-traffic and will severely degrade performance. In the WB approach, the updated counter block is marked as dirty in the counter cache and is only written back to NVM when it is evicted. The main issue with the WB counter-cache is that it requires a sufficient and reliable power source after a crash to enable flushing the counter-cache contents to NVM, which increases the system cost and typically requires a large area. Most processor vendors, e.g., Intel, define the in-processor persistent domain to only include the write-pending queue (WPQ) in the memory controller. Note that using UPS to hold-up the system until backing up important data is a common practice in critical systems. However, we do not expect commodity or low-power processors to shoulder the costs and area requirements of UPS systems.

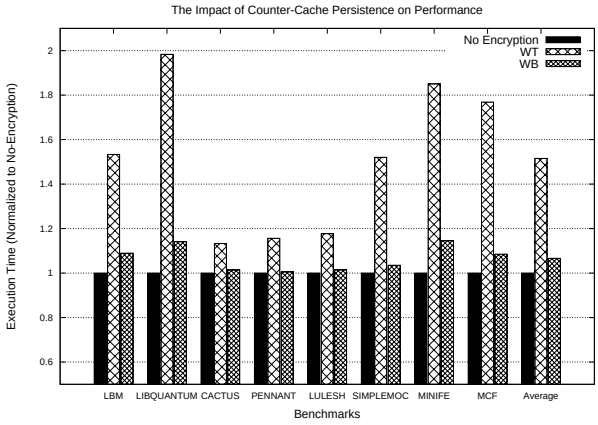

Fig. 4: Impact of counter-cache persistence on performance.

Shown in Figure 4, WT counter-cache entails significant performance degradation for most of the benchmarks compared to no-encryption and WB counter-cache schemes. On average, WT scheme degrades performance by 51.5% whereas WB scheme only degrades performance by 6.6% compared to a no-encryption (unprotected) scheme. In applications that are write-intensive, e.g., `libquantum` benchmark, the performance overhead of WT can reach 198% normalized to that of the no-encryption scheme.

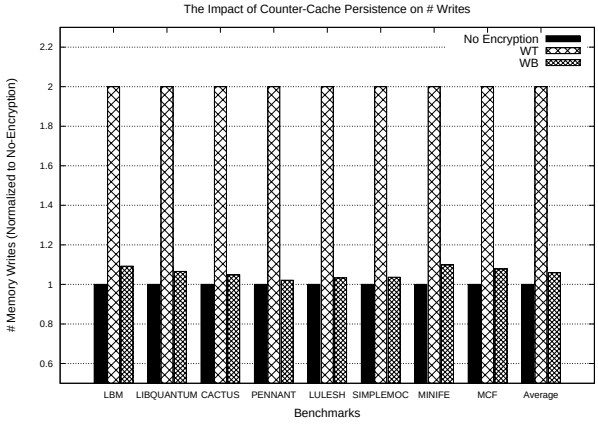

Fig. 5: Impact of counter-cache persistence on number of writes.

Another key metric relevant to NVM is the write traffic. NVMs have limited write bandwidth due to the limited write-drivers and the large power-consumption for each write operation. More critically, NVMs have limited endurance for

writes. In WT scheme, besides data block write, each write operation requires an additional in-memory persistence of the counter value, thus incurs twice the write traffic of the no-encryption scheme. In contrast, the WB scheme only writes the updated counter blocks to the NVM when blocks are evicted from the counter cache. As shown in Figure 5, WB incurs an extra 5.92% writes on average when compared to the no-encryption scheme, whereas WT, as expected, incurs twice the write traffic.

Note that the *atomicity* of the WT scheme includes both data write and counter write. Crashes (or power loss) can happen between transactions or in the middle of a transaction. There are several simple ways to guarantee atomicity. ① Placing a small NVM storage (e.g., 128B) inside the processor chip to log both the to-be-written data block and counter block before trying to commit them to memory. If a failure occurs, once the system restores, it begins with redoing the stored transactions. ② Guaranteeing enough power hold-up time (through reasonable small capacitors) to complete at least 2 write operations (data and counter). However, solutions that aim to guarantee memory transaction-level atomicity are beyond the scope of this paper.

In this paper, we propose a new scheme that has the advantages of the WT scheme — no need for a battery or long power hold-up time. It also has as small performance and write traffic overheads such as those of the WB scheme.

III. Osiris

In this section, we first discuss our assumed threat model. Then, we discuss the design of Osiris and the possible design options and trade-offs.

A. Threat Model

Our assumed threat model is similar to state-of-the-art work on secure processors [4], [6], [8]. The trust base includes the processor and all of its internal structures. Our threat model assumes that an attacker can snoop the memory bus, scan the memory content, tamper with the memory content (including rowhammer) and replay old packets. Differential power attacks and electromagnetic inference attacks are beyond the scope of this paper. Furthermore, attacks that try to exploit processor bugs in speculative execution, such as Meltdown and Spectre, are beyond the scope of this paper. Such attacks can be mitigated through more aggressive memory fencing around critical code to prevent speculative execution. Finally, our proposed solution does not preclude secure enclaves and hence can operate in untrusted Operating System (OS) environments.

Attack on Reusing Counter Values for Non-Persistent Data: While state-of-the-art [13] work relaxes persisting counters for non-persistent data, if using split counter like in [15], it actually introduces serious security vulnerabilities. For example, assume an attacker application uses known-plaintext and writes it to memory. If the memory location is non-persistent, the encrypted data will be written to memory but not the counter. Thus, by observing the memory bus, the attacker can discover the encryption pad by XORing the observed ciphertext, $(E_{key}(IV_{new}) \oplus Plaintext)$, with the $Plaintext$. Note that it is also easy to predict the plaintext for some accesses, for instance, zeroing at first access. By now, the attacker knows the value of $E_{key}(IV_{new})$. Later, after a crash, the memory controller will read IV_{old} and increment it, which generates IV_{new}, and then encrypts the new application data written to that location to become $E_{key}(IV_{new}) \oplus Plaintext2$. The attacker can discover the value of $Plaintext2$ by XORing the ciphertext with the previously observed $E_{key}(IV_{new})$. Note that the stale counter could have been already incremented multiple times before the crash; multiple writes of the new application can reuse counters with known encryption pads. Note that such an attack only needs a malicious application to run (or just predictable initial plaintext of an application) and a physical attacker or bus snooper.

B. Design Options

Before delving deep into the details of Osiris, let's first discuss the challenges of securely recovering the encryption counters after a crash. Without a mechanism to persist encryption counters, once a crash occurs, you are only guaranteed that the root (kept in processor) of the Merkle Tree is updated and reflects the most recent counter values written to memory; any write operation before being sent to NVM will update the affected parts/branches of the Merkle Tree up to the root. Note that it is likely that the affected branches of the Merkle Tree will be cached on the processor and there is no need to persist them as long as the processor can maintain the value of the root after crash. Later, once the power is back and we want to restore the system, we may have stale counter values in the NVM and stale intermediate values of the Merkle Tree.

Once the system is powered back, any access to a memory location needs two main steps: ① obtain the corresponding most-recent encryption counter from memory and ② verify the integrity of data through the MAC and the integrity of the used counter value through the Merkle Tree. It is possible that Step ① results in a stale counter, in which case Step ② will fail due to a Merkle Tree mismatch. Remember that the root of the Merkle Tree was updated before the crash, thus using a stale counter will be detected. As soon as the error is detected, the recovery process stops. One naïve approach would be to try all possible counter values and use the Merkle Tree to verify each value. Unfortunately, such a brute-force approach is impractical for several reasons. First, finding out the actual value requires trying all possible values for a counter paired with calculating the corresponding hash values to verify integrity, which is impractical for typical encryption counter schemes where there could be 2^{64} possible values for each counter. Second, it is unlikely that only one counter value is stale; many updated counters in the counter cache will be lost. Thus, reconstructing the Merkle Tree will be impractical if there are multiple stale counters. For example, counters of blocks X and Y are lost, then we need to reconstruct the Merkle Tree with all possible combinations/values of X and Y and then compare the resulting root with the one safely stored in processor. For simplicity we only mention losing two counters; in a real system, where a counter cache is hundreds of kilobytes, there will likely be thousands of stale blocks.

Observation 1: Losing encryption counter values makes reconstructing the Merkle Tree nearly impossible. Approaches such as a brute-force trial of all possibly lost counter values

to reconstruct the tree will take an impractical amount of time especially when multiple counter values have been lost.

One possible way to reduce reconstruction time is to employ ***stop-loss*** mechanisms that limit the number of possible counter values to verify for each counter after recovery. Unfortunately, since there is no way to pinpoint exactly which counters have lost their values, an aggressive search mechanism is still needed. If we limit the number of writes to each counter block before persisting it to only N, then we need to try up to N^S combinations for reconstruction where S is the number of data blocks. For instance, let's assume we have a 128GB NVM memory and 64B cache blocks, then we have 2G blocks. If we only set N to 2, then we need up to $2^{2^{31}} = 2^{2147483648}$ trials. Accordingly, a stop-loss mechanism could reduce the time to reconstruct the Merkle Tree. However, it is still impractical.

Obviously, a more explicit confirmation is needed before proceeding with an arbitrary counter value to reconstruct the Merkle Tree. In other words, we need a hint on what the most recent counter value was for each counter block. For instance, if the previously discussed stop-loss mechanism is used along with an approach to bookkeep the *phase* within the N trials before writing the whole counter block, then we can start with a more educated guess. Specifically, each time we update a counter block N times in the counter-cache, we need to persist its Nth update in the memory, which means that we need $log_2 N$ bits (i.e., phase) for each counter block to be written atomically with the data block. Later, when the system starts recovery, it knows the exact difference between the most recent counter value and the one used to encrypt the data through the phase value.

Observation 2: Co-locating the data blocks with a few bits that reflect most-recent counter value used for encryption can enable fast-recovery of the counter-value used for encryption. Note that if an attacker tries to replay old data along with their phase bits, then the Merkle Tree verification will detect the tampering due to a mismatch in the resulting root of the Merkle Tree.

Although stop-loss along with phase storage can make the recovery time practical, adding more bits in memory for each cache line is tricky for several reasons. First, as discussed in [13], increasing the bus-width requires adding more pins to the processor. Even avoiding extra pins by adding extra burst in addition to the 8 bursts of 64-bit bus width for each 64B block is expensive and requires support from DIMMs in addition to underutilization of data bus (only few bits are written in the 64-bit wide memory bus in the last burst). Second, major memory organization changes are needed, e.g., row-buffer size, memory controller timing and DIMM support. Additionally, cache blocks are no longer 64B aligned, which can cause complexity in addressing. Finally, extra bit writes are needed for each cache line to reflect the counter phase, which can map to a different memory bank, hence additional occupation of bank for write latency.

To retain the advantages of stop-loss paired with phase bookkeeping but without extra bits, Osiris repurposes already existing ECC bits as a fast counter recovery mechanism. The following subsection will discuss in details how Osiris can elegantly employ ECC bits of data to find out the counter used for encryption.

C. Design

Osiris mainly relies on inferring the correctness of an encryption counter by calculating the ECC associated with the decrypted text and comparing it with that encrypted and stored with the encrypted cacheline. In conventional (not encrypted) systems, when the memory controller writes a cache block to the memory it also calculates its ECC, e.g., hamming code, and stores it with the cacheline. In other words, the tuple that will be written to the memory when writing cache block X to memory is $\{X, ECC(X)\}$. In contrast, in encrypted memory systems, there are two options to calculate ECC: ① using the plaintext, then encrypting it with the cacheline before writing both to memory or ② encrypting the plaintext, then calculating the ECC over the encrypted block before both are written to memory. Although approach ② allows overlapping decryption and ECC verification, most ECC implementations used in memory controllers, e.g., SEC-DED Hsiao Code [16], take less than a nanosecond to complete [17]–[19], which is negligible compared to cache or memory access latencies [20]. Additionally, pipelining the arrival of bursts with decryption and ECC bits decoding will completely hide the latency. However, we observe that calculating ECC bits over the plaintext and encrypting it along with the cacheline can provide low-cost and fast way of verifying the correctness of the encryption/decryption operation. Specifically, in counter-mode encryption, the data is decrypted using the following: $\{X, Z\} = E_{key}(IV_X) \oplus Y$, where Y is supposedly/potentially the encryption of X along with its ECC and Z is potentially equal to ECC(X). In conventional systems, if $ECC(X) \neq Z$, then the reason is definitely an error (or tampering) occurred on X or Z. However, when counter-mode encryption is used, the reason could be that an error (or tampering) occurred on X or Z, or the *wrong* IV is used to do the encryption, i.e., decryption is not done successfully.

Observation 3: When the ECC function is applied over the plaintext and the resulting ECC bits are encrypted along with the data, the ECC bits can provide a sanity-check for the encryption counter. Any tampering with the counter value will be detected by a *clear* mismatch of the ECC bits that result from that invalid decryption; results of $E_{key}(IV_{old})$ and $E_{key}(IV_{new})$ are very different and independent. Note that in a Bonsai Merkle Tree, data-integrity is protected through MAC values that are calculated over each data and its corresponding counter. While relying on ECC for sanity-checking, the ECC bits can fail to provide guarantees as strong as cryptographic MAC values. Accordingly, we adopt Bonsai Merkle Trees to add additional protection for data integrity. However, ECC bits, when combined with counter-verification mechanisms, can provide tamper-resistance as strong as the error detection of the used ECC algorithm.

Important Note: Stream ciphers, e.g., CTR and OFP modes, do not propagate errors, i.e., an error in ith bit of encrypted data will result in an error in ith bit of decrypted data, hence the reliability is not affected. In encryption modes where an error in encrypted data can result in completely unrelated decrypted

text, e.g., block cipher modes, careful consideration is required as encrypting ECC bits can render them useless when there is an error. For our scheme, we focus on state-of-the-art memory encryption, which commonly uses CTR-mode for security and performance reasons.

Now the question is how to proceed when there is an error detected due to a mismatch between the expected ECC and the stored ECC (obtained after decryption). As the reader would expect, the first step is to find if the error is correctable using the ECC code. Table I lists some common ECC errors and their associated fixes.

TABLE I: Common Sources of ECC Mismatch

Error Type	Common fix
Error on stored data	Can be fixed if the error is correctable, e.g. single bit failure
Error on ECC	Typically unrecoverable
Stale/Wrong IV	Speculate the correct IV and verify

As shown in Table I, one reason for such an ECC mismatch is using an old IV value. To better understand how this can happen, recall the counter cache persistence issue. If a cache block is updated in memory, it is also necessary to update and persist its encryption counter, for both security and correctness reasons. Given the ability to detect the use of stale counter/IV, we can *implicitly* reason about the likelihood of losing the most-recent counter value due to a sudden power loss. To that extent, in theory, we can try to decrypt the data with all possible IVs and stop when an IV successfully decrypts the block, i.e., the resulting ECC matches the expected one ($ECC(X) = Z$). At that point, there is a high chance that the IV was actually the one used to encrypt the block, but it was either lost due to inability to persist the new counter value after persisting the data, or due to a *relaxed scheme*. Osiris builds upon the later possibility; it uses a relaxed counter persistence scheme to employ ECC bits to verify the correctness of the counter used for encryption. As discussed earlier, it is impractical to try all the possible IVs to infer the IV used to encrypt the block, thus Osiris deploys the stop-loss mechanism to limit such possibility to only N updates of a counter, i.e., the correct IV should be within $[IV+1, IV+N]$, where IV is the most recent IV that was stored/persisted in memory. Note that once the speculated/chosen IV passes the first check through ECC sanity-check it also needs to be verified through Merkle Tree.

We propose two flavors for Osiris, baseline ***Osiris*** and ***Osiris-Plus***. In baseline Osiris, during normal operation, all counters being read from memory reflect their most-recent values; the most-recent value is either updated in cache or has been evicted/written-back to memory. Thus, inconsistency between counters in memory and counters in cache can happen due to crash or tampering with counters in memory. In contrast, Osiris-Plus strives to even outperform Battery-Backed Write-Back counter-cache scheme through purposely skipping counter updates and recovering their most-recent values when reading them back, hence inconsistency can happen during normal operation. Below is a further discussion on both baseline Osiris and Osiris-Plus.

Normal Operation Stage: During normal operation, Osiris adopts a write-back mechanism by updating memory counters only when evicted from the counter cache. Thus, Osiris can always find the most-recent counter value either in cache or by fetching it from memory in the case of miss. Accordingly, in normal operation, Osiris is similar to conventional memory encryption except that a counter is persisted once each Nth update — acting like a write-through for the Nth update of each counter. In contrast, Osiris-Plus allows occasional dropping of the most-recent values of encryption counters by relying on run-time recovery mechanisms of the most-recent values of counters. In other words, Osiris-Plus relies on trying multiple possible values on each counter miss to recover the most-recent one before verifying it through a Merkle Tree; baseline Osiris only does this at system recovery time.

System Recovery Stage: During system recovery time, both Osiris and Osiris-Plus must reconstruct the Merkle Tree at the time of restoration. Furthermore, both need to use the most-recent values of counters to reconstruct a tree that has a root that matches the root stored in the secure processor. In both Osiris and Osiris-Plus, the system recovery will start by traversing all memory locations (cache lines) in an integrity verified region (all memory in secure NVM). For each cache line location, i.e., 64B address, the memory controller uses the ECC value after decryption as a sanity check of the counter retrieved from memory. However, if the counter value fails the check, all possible N values must be checked to find the most-recently used counter value. Later, the correct value will overwrite the current (stale value) counter in memory. After all memory locations are checked, the Merkle Tree will be reconstructed with the recovered counter values and, eventually, build up all intermediate nodes and the root. In the final step, the resulting root will be compared with that saved and kept in the processor. If a mismatch occurs, the data integrity of the memory cannot be verified and it is very likely that an attacker has replayed both counter and corresponding encrypted data+ECC blocks.

Next, we will discuss the design of Osiris and Osiris+Plus by guiding the reader through the steps of read and write operations in both schemes.

1) Osiris Read and Write Operations: As shown in Figure 6, during a write operation, once a cache block is evicted from LLC, the memory controller will calculate the ECC of the data in the block, as shown in Step ①. Note that write operations happen in the background and are typically buffered in the write-pending queue. In Step ②, the memory controller obtains the corresponding counter in the event of a miss and evict/write-back the evicted counter block, if dirty. The counter value obtained from Step ② will be used to proactively generate the encryption pad, as shown in Step ③. In Step ④, the counter value will be verified (in case of miss) and the counter and affected Merkle Tree (including the root node) are updated in the Merkle Tree cache. Unlike a typical write-back counter-cache, if the new counter value is a multiple of N or 0, then the

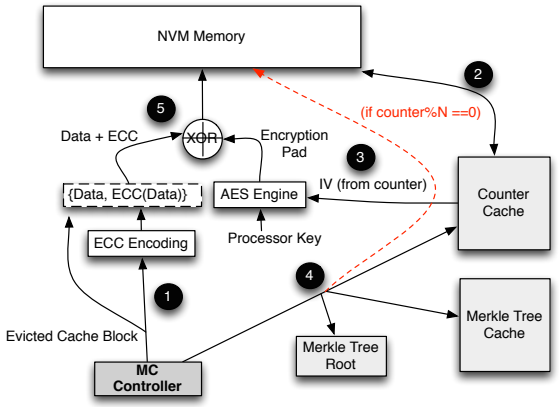

Fig. 6: Osiris Write Operation

new counter value is also persisted before proceeding. Finally, in Step ⑤, the data+ECC is encrypted using the encryption pad and written to memory.

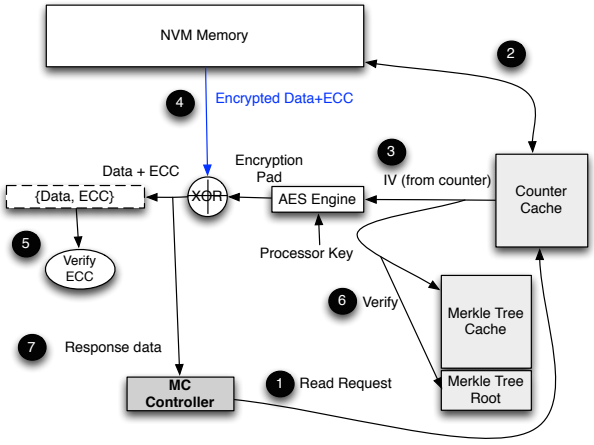

Fig. 7: Osiris Read Operation

During a read operation, as shown in Figure 7, the memory controller will obtain the corresponding counter value from the counter cache (in the case of hit) or memory (in the case of a miss) and evict the victim block, if dirty, as shown in Steps ① and ②. The counter value will be used to generate the encryption pad as shown in Step ③. In Step ④, the actual data block is read from memory and decrypted using the pad generated in Step ③. In Step ⑤, traditional ECC checking occurs. Finally, if the counter block is fetched from memory (miss), the integrity of the counter value is verified, as shown in Step ⑥. Finally, as shown in Step ⑦, the memory controller receives the decrypted data, which is then forwarded to the cache hierarchy by the memory controller. Note that many of the steps can be overlapped without any conflicts, for instance, Step ⑥ and steps ④ and ⑤.

2) Osiris-Plus Read and Write Operations: The write operation in Osiris-Plus is very similar to the write-operation in baseline Osiris; the difference is that it does not write back evicted dirty blocks from the counter cache, as could happen in Step ② of Figure 6. Instead, Osiris-Plus recovers the most-recent value of the counter each-time it is read from memory and only updates it in memory each Nth update. Figure 8 depicts the read operation of Osiris-Plus.

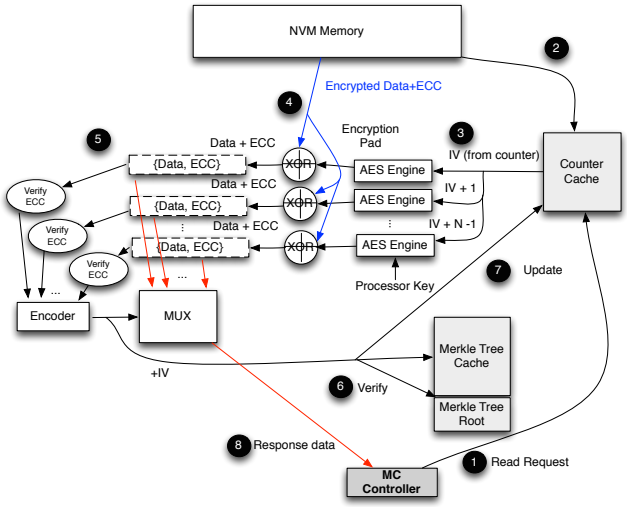

Fig. 8: Osiris-Plus Read Operation

The main difference between Osiris and Osiris-Plus is shown in Steps ⑤ and ⑥ in Figure 8. Osiris-Plus utilizes additional encryption engines and ECC units to allow fast recovery of the most-recent counter value. Note that given the fact that most AES encryption engines are pipelined and the candidate counter values are sequential, using a single encryption engine instead of N engines can only add N extra cycles. Once a counter value is detected, through post-decryption ECC verification, to be the most-recent one, it will be verified through a Merkle Tree similar to baseline Osiris. Once the counter is verified, the data resulting from decrypting the ciphertext with the most-recent counter value will be returned to the memory controller. Note that the recovered counter value is updated in the counter-cache, as shown in Step ⑦.

D. Reliability and Recovery from Crash

As mentioned earlier, to recover from a crash, Osiris and Osiris-Plus must first recover the most-recent counter values utilizing the post-decryption ECC bits to find the correct counters. The Merkle Tree will then be constructed and the resulting root will be verified and compared with the root kept in the processor. While the process seems reasonably simple, it can get complicated in the presence of errors. Specifically, uncorrectable errors in the data or encryption counters make generating a matching Merkle Tree root impossible, which means that the system is unable to verify the integrity of memory. Note that such uncorrectable errors will have the same effect on integrity-verified memories even without deploying Osiris.

Uncorrectable errors in encryption counters protected by a Merkle Tree can potentially fail the recovery process. Specifically, when the system attempts to reconstruct the Merkle Tree, it will fail to generate a matching root. Furthermore, it is not feasible to know which part of the Merkle Tree causes the problem. Only the root is maintained; all other parts of the tree should generate the same root or none should be trusted.

One way to mitigate this single-point of failure is to keep other parts of the Merkle Tree in the processor and guarantee they are never lost. For instance, for an 8-ary Merkle Tree, the 8 immediate children of the root are also saved. In a more capable system, the root, its 8 immediate children and their children are kept — a total of 73 MAC values. If the recovery process fails to produce the root of the Merkle Tree, we can look at which children are mismatched and declare that part of the tree as un-verifiable and warn the user/OS. While we provide insight into this problem, we assume the system architects choose to only save the root. However, having more NVM registers inside the processors to save more levels of the Merkle Tree can be implemented for high error systems. We leave reconstructing a Merkle Tree in the presence of uncorrectable errors as future work.

To formally describe our recovery process success rate, we can look at two cases. In the first case, when no errors occur in the data, since each 64B memory block has 8B ECC (64 bits), the probability that a wrong counter results in correct (indicate no errors) ECC bits for the whole block is only $\frac{1}{2^{64}}$, i.e., similar probability to guessing a 64-bit key correctly, which is next to impossible. Note that each 64B block is practically divided into 8 64-bit words [21], each has its own 8-bit ECC, i.e., each bus burst will have 64-bit data and 8-bit ECC (total of 72 bits). This is why most ECC-enabled memory systems will have 9 chips per rank instead of 8, where each chip provides 8 bits. The 9th chip provides the ECC 8-bits for the burst/word. Also note that the 64B block will be read/written as 8 bursts on a 72-bit memory bus.

Bearing in mind the organization of ECC codewords for each 64B memory block, let's now discuss the case where there is actually an error in data. For an incorrect counter, the probability that an 8-bit ECC codeword indicates that there is no error is $P_{no-error} = \frac{1}{2^8}$ and the probability of not indicating that there is no-error, i.e., that there is an error, is $P_{error} = 1 - \frac{1}{2^8}$. The probability that k codewords of the 8 ECC codewords indicate an error can be represented by a Bernoulli Trial as $P_k = \binom{8}{k} \times (1 - \frac{1}{2^8})^K \times (\frac{1}{2^8})^{8-K}$. For example, P_2 represents the probability of having 2 of the 8 ECC codewords flagging an error for a wrongly decrypted data and ECC (semi-randomly generated). Accordingly, if we use our metric to filter encryption counters that have 4 or more ECC codewords indicating an error, then the probability of our success in detecting wrong counters can be given by $P_{k\geq 4} = 1 - (P_0 + P_1 + P_2 + P_3)$. We find that $P_{k\geq 4}$ is nearly 100% (improbability less than 5.04×10^{-11}). Thus, by filtering out counters cause 4 or more incorrect ECC codewords, we can with a probability of almost 100% detect any wrong counter. However, if the data has 4 or more (out of 8) words with at least an error on each of them, then also the correct counter will be filtered out and Merkle Tree verification will be used with each of the possible counter values to identify the correct counter before correcting data. In other words, Osiris will reliably filter out wrong counters immediately except for the rare case of having at least 4 words each with at least an error in the same data block. However, since in case of Osiris only few thousands of blocks (those were in cache) are stale at recovery time, the probability that one of them corresponds to a data block that has 4 or more faulty words is very small, which then would cause an additional step to find the correct counter. Note that Osiris does not affect reliability but rather incurs additional step for identifying the correct counter when the original data has large number of faulty words.

It is also important to note that many PCM prototypes use similar ECC organization but with larger number of ECC bits per 64-bit words, e.g., 16-bit ECC for each 64-bit word [22], which makes our detection success rate even closer to perfect. Specifically, using Bernoulli trial for 16-bit ECC per word, the probability of a wrong counter causes 5 or more mismatching ECC codewords is almost 100% (improbability less than 2×10^{-13}), and thus only blocks with at least 5 (out of 8) words are faulty would require additional Merkle Tree verification to recognize the correct counter. Finally, in the rare case that a wrong counter is falsely not filtered out, Osiris will rely on Merkle Tree to identify the correct counter from those not filtered out.

E. Systems without ECC

Instead of ECC bits, some systems employ MAC values that can be co-located with data [8]. For instance, the ECC chips in the DIMM can be used to store the MAC values of the Bonsai Merkle Tree, allowing the system to obtain data integrity-verification MACs along with the data. While our description of Osiris and Osiris-Plus focuses on using ECC, MAC values can also be used for the same purpose — as a sanity-check for the decryption counter. The only difference is when an error is detected. With ECC, the number of mismatched bits can be used to guess the counter value. This isn't true for MAC values, which tend to differ significantly in the presence of an error. To solve this problem, MAC values that are aggregations of multiple MAC-Per-Word, e.g., 64 bits that are made of 8 bits for each 64 data bits, are used. The difference between the generated MACs and the stored MACs for different counter values can be used as a selection criteria for the candidate counter value. Our proposed schemes also work with ECC and MAC co-designs such as Synergy (parity+MAC) [8].

F. Security of Osiris and Osiris-Plus

Since Osiris and Osiris-Plus use Merkle Trees for final verification, they have security guarantees and tamper-resistance similar to any scheme that uses a Bonsai Merkle Tree, such as state-of-the-art secure memory encryption schemes [4], [6], [8].

IV. Evaluation

In this section, we evaluate the performance of Osiris with other state-of-the-art persisting cache designs. Section IV-A describes the methodology we use for the experiments. Section IV-B discusses the performance impact of the design for Osiris/Osiris Plus in terms of limit value and the performance comparison with other cache designs.

TABLE II: Configuration of the simulated system.

Processor	
CPU	4-core, 1GHz, out-of-order x86-64
L1 Cache	private, 2 cycles, 32KB, 2-way, 64B block
L2 Cache	private, 20 cycles, 512KB, 8-way, 64B block
L3 Cache	shared, 32 cycles, 8MB, 64-way, 64B block
DDR-based PCM Main Memory	
Capacity	16 GB
PCM Latencies	60ns read, 150ns write [23]
Organization	2 ranks/channel, 8 banks/rank, 1KB row buffer, Open Adaptive page policy, RoRaBaChCo address mapping
DDR Timing	tRCD 55ns, tXAW 50ns, tBURST 5ns, tWR 150ns, tRFC 5ns [13], [23] tCL 12.5ns, 64-bit bus width, 1200 MHz Clock
Encryption Parameters	
Counter Cache	256KB, 16-way, 64B block

A. Methodology

We model Osiris in Gem5 [24] with the system configuration presented in the Table II. We implement a 256KB, 16-way set associative counter cache, with a total number of 4K counters. To stress-test our design, we select memory-intensive applications. Specifically, we select eight memory-intensive representative benchmarks from the SPEC2006 suite [25] and from proxy applications provided by the U.S. Department of Energy (DoE) [26]. The goal is to evaluate the performance and the write traffic overheads of our design. In all experiments, the applications are fast-forwarded to skip the initialization phase, and then followed by the simulation of 500 million instructions. Similar to prior work [9], we assume the overall AES encryption latency to be 24 cycles, and we overlap fetching data with encryption pad generation. Below are the schemes we use in our evaluation:

No-encryption Scheme: The baseline NVM system without memory encryption.

Osiris Scheme: Our base solution that eliminates the need for a battery while minimizing the performance and write traffic overheads.

Osiris Plus Scheme: An optimized version of the Osiris scheme that also eliminates the need for evicting dirty counter blocks at the cost of extra online checking to recover the most recent counter value.

Write-through (WT) Scheme: A strict counter-atomicity scheme that, for each write operation, enforces both counter and data blocks to be written to NVM memory. Note that this scheme does not require any unrealistic battery or power supply hold-up time.

Write-back (WB) Scheme: A battery-backed counter cache scheme. The WB scheme only writes to memory dirty evictions from counter cache. However, the WB scheme assumes a battery is sufficient to flush all dirty blocks in counter cache.

B. Analysis

As discussed in Section III-C, the purpose of Osiris/Osiris-Plus is to persist the encryption counters in NVM memory in response to system crash recovery but with reduced performance overhead and write traffic. Therefore, the selection of number *N* for update interval (also discussed in Section III-C) is critical in determining the performance improvement. As such, in this section, we study the impact of choosing different *N* (limit) for Osiris/Osris-Plus on performance. Next, we present the performance analysis of multiple benchmarks in response to different persistent schemes discussed and compared in this paper. Our evaluation results are consistent with the goal of our design for Osiris and Osiris-Plus.

1) Impact of Osiris Limit: To understand the impact of Osiris limit (also called N) on performance and write traffic, we vary the limit in multiples of two and observe the corresponding performance and write traffic. Figure 9 shows the performance of Osiris and Osiris-Plus normalized to no-encryption while varying the limit. The figure also shows WB and WT performance normalized to no-encryption to facilitate the comparison.

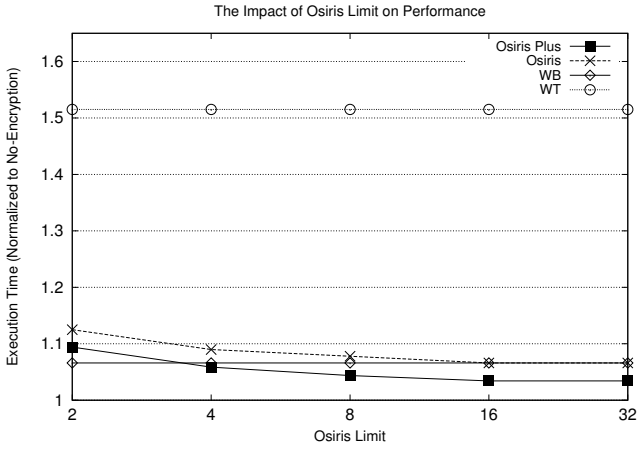

Fig. 9: The impact of Osiris limit on performance

From Figure 9, we can observe that both Osiris and Osiris-Plus benefit clearly from increasing the limit. However, as discussed earlier, having large limit values can cause increase in recovery time and potentially large number of encryption engines and ECC units (in case of Osiris-Plus). Accordingly, it is important to find the point which can no longer bring in justifiable gains if increased. From Figure 9, we can observe that Osiris at limit 4 has an average performance overhead of 8.9% compared to 6.6% and 51.5% for WB and WT, respectively. In contrast, also at limit 4, Osiris-Plus has only 5.8% performance overhead, which is even better than WB scheme. Accordingly, we can observe that at limit 4, both Osiris and Osiris-Plus perform close to or outperform the WB scheme even though they do not need any battery or hold-up power. In large limit values, e.g., 32, Osiris performs similar to write-back; dirty blocks will be evicted before absorbing the limit of number of writes, hence the counter block is rarely persisted before eviction. In contrast, Osiris-Plus brings down the performance overheads to only 3.4% (compared to 6.6% for WB), but again at the cost of large number of encryption engines and ECC units.

A similar pattern can be observed in Figure 10 demonstrating less write traffic for both schemes. At limit 4, Osiris has a write traffic overhead of 8.5% whereas WB and WT have 5.9% and 100%, respectively. In contrast, at limit 4, Osiris-Plus has only 2.6% write traffic overhead. With larger limit values, e.g., 32, Osiris-Plus has nearly zero extra writes, whereas Osiris has write traffic overhead similar to WB. Based on this observation, we use limit 4 as a reasonable trade-off between

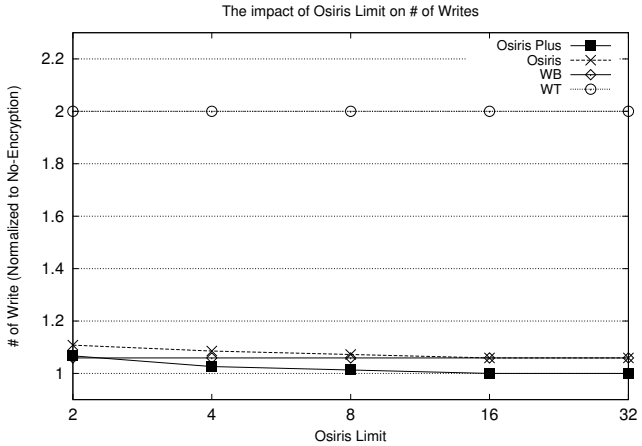

Fig. 10: The impact of Osiris limit on number of writes.

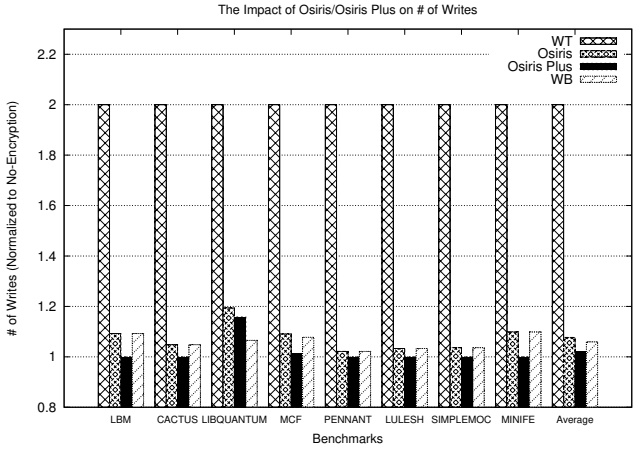

Fig. 12: The impact of Osiris and Osiris Plus persistence on number of writes

the performance overhead and the required additional stages or extra checking at the time of recovery.

2) Impact of Osiris and Osiris Plus persistency on multiple benchmarks: As we now understand the overall impact of Osiris-limit, i.e., N, on performance and write traffic, we now zoom-in on the performance and write traffic of individual benchmarks when using limit 4 as suggested in Section IV-B1.

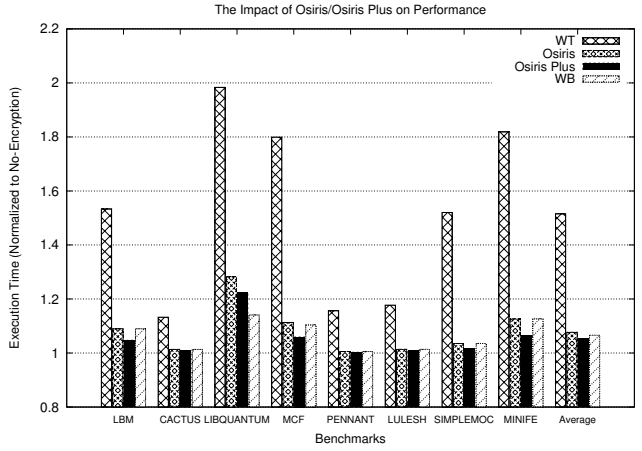

Fig. 11: The impact of Osiris and Osiris Plus persistence on performance

As we can observe from Figure 11 and Figure 12, for most of the benchmarks, Osiris-Plus outperforms all other schemes in both performance and reduction in number of writes. Meanwhile, for most benchmarks, Osiris performs close to WB scheme. As noted earlier, Osiris performance and write traffic are bounded by the WB scheme; if the updated counters rarely get persisted before eviction from counter cache, i.e., updated less than N time, then Osiris performs similar to WB but without need for battery. Note that it is not common to have the same counter updated many times before eviction from counter cache; once a data cache block gets evicted from the cache hierarchy, it is very unlikely that it will be evicted again very soon. However, since Osiris-Plus can additionally eliminate premature (before Nth write) counter evictions, it can actually outperform WB. The only exception we can observe is `Libquantum`, which is mainly due to its behavior of repeated streaming behavior over a small array (4MB array); many cache hierarchy evictions due to conflicts, however, since each counter cache block covers the counters of 4KB page, many evictions/writes of the same data block will result in hits in the counter cache. Accordingly, a WB scheme performs better as there are few evictions (write-backs) from counter cache compared to the writes due to persistence of counter values at each Nth write in Osiris and Osiris-Plus. However, we observe that with larger limit value, such as 16, Osiris-Plus clearly outperforms WB (7.8% vs. 14% overhead), whereas Osiris performs similarly. In summary, for all benchmarks, Osiris and Osiris-Plus performs better than strict-counter persistence (WT) and very close or even better than battery-backed WB scheme. In addition, we also tested some aggressive cases, such as read latency 300ns and write latency 1000ns. On average, the Osiris-Plus outperforms WB in both execution overhead and number of writes; Osiris, although slightly degraded, is still very close to WB scheme's performance.

3) Sensitivity to Counter Cache Size: While it is impractical to assume a very large battery-backed write-back cache, we verify the robustness of Osiris and Osiris-Plus when large counter caches are used. Even at 1MB cache size, Osiris and Osiris-Plus have performance overhead of only 3.8% and 3%, whereas WT and WB has 48.9% and 1.4%, respectively. Similarly, Osiris and Osiris-Plus have write traffic overhead of only 3.8% and 2.6%, whereas WT and WB has 100% and 1.2%, respectively. Note that WB is only slightly better due to the long tenure of cache blocks before eviction. Due to the limited space, we do not include different cache sizes between 256KB and 1MB, however, the trend is as expected and always in favor of Osiris and Osiris-Plus.

4) Recovery Time: At recovery time in a conventional NVM system, the Merkle Tree must be first reconstructed as early as the first memory access. The process will build up the intermediate nodes and the root to compare it with that kept inside the processor. To do so, it will also need to verify the counters' and data integrity through comparison with the MAC calculated over the counter and data.

Baseline Osiris, in the worst case scenario of a crash,

probably have lost all the updated counter cache blocks, i.e., 2048 64B counter blocks for a 128KB counter cache. During recovery, the memory controller will iterate over all encryption counters and build the Merkle Tree. However for counters that have a mismatched ECC, an average of 4 trials (when N equals 8) of counter values will be tried before finding the correct value. Thus, for a 16GB NVM memory, there will be 256M data blocks and 4M counter blocks. Assuming that reading a data block and verifying its counter takes $100ns$, then in a conventional system we need $100ns \times 256M$ which is roughly 25.6 seconds.

In Osiris, only 131072 counters (corresponding to the lost 2048 counter blocks) will require additional checking. In the worst case, each counter has been updated 8 times, which will need $131072 \times 8 \times 100ns$ extra delay, which extends the recovery time to 25.7 seconds – only an additional 0.4% recovery time. Also note that Osiris overhead depends on the counter cache size. Using large NVM memory capacity will significantly reduce the overhead percentage due to an increase in actual recovery time.

In contrast, as mentioned in the paper, Osiris-Plus deploys parallel engines that check all possible values in parallel, which means that even for wrong values the detection and verification latency will be similar to those of the correct counters and the correct counter values will be used to build upper intermediate levels. Accordingly, there is no additional overhead in the recovery time for Osiris-Plus. However, this requires extra parallel ECC and AES engines. Note that most modern processors have multiple memory channels and a high degree of parallelism in memory modules, which makes the cost of $100ns$ to sequentially access and verify each block be conservative. We estimate recovery time to be even faster for both conventional systems and Osiris.

V. Related Work

In this section, we review and discuss the relevant work in NVM security, mostly in respect to encryption and memory persistence in this section.

NVM Security: Most research advocates counter-mode encryption (CME) to provide strong security while benefiting from short latency due to the overlap of encryption-pad generation with data fetch from memory [4], [6], [13], [27]–[30]. Further optimization is carried out based on this fundamental model. DEUCE [6] proposes a dual-counter CME scheme that only re-encrypts the modified word for each write. SECRET [27] integrates zero-based partial writes with XOR-based energy masking to reduce both overhead and power consumption for encryption. ASSURE [28] further eliminates cell writes of SECRET by enabling partial MAC computation and constructing efficient Merkle Tree. Distinctively, Silent Shredder [4] repurposes the IVs in CME to eliminate the data shredding writes. More recently, Synergy [8] described a security-reliability co-design that co-locates MAC and data for data reconstruction due to single-chip error and for reducing the overall memory traffic. In contrast with prior work, our work is the first to provide a security-reliability guaranteed solution for persisting encryption counters for all memory blocks.

Memory Persistence: As a new technology possessing both the properties of main memory and storage, persistent memory has promising recovery capabilities and the potential to replace main memory. Thus it attracts much attention from both hardware and software research communities [31]–[34]. Pelley et al. [35] refer to memory persistency as the constraints of write ordering with respect to failure and propose several persistency models in either a strict or a relaxed way. DPO [36] and WHISPER [37] are different persistent frameworks that use strict and relaxed ordering, respectively, with different granularities. In our paper, we assume a conventional persistence model, i.e., cacheline flushing ordered by store-fencing, e.g., `CLWB` then `SFENCE`, to persist memory locations [38] and we focus on the persistence from a hardware perspective.

The most related work to Osiris is counter-atomicity [13]. Counter-atomicity introduces a counter write queue and a ready bit to enforce write persistence of data and counters from the same request to NVM simultaneously. To reduce the write overhead, counter-atomicity provides some APIs for users to selectively persist self-defined critical data along with their counters, relaxing the need for all-counter persistence. Osiris tackles the same problem but by persisting counters periodically in encrypted NVMs. Osiris relies on ECC bits to provide sanity-check for the used encryption counter and helps with its recovery. While our solution does not require any software modification and is completely transparent to users, it is orthogonal to and can be augmented with selective counter-atomicity scheme. Moreover, Osiris and Osiris-Plus can be used in addition to selective counter-atomicity to cover all memory locations at low-cost, hence avoiding known-plaintext attacks discussed earlier in Section III-A. The work discussed early in this paper consider failure as a broad concept [35], whereas some work specifically handle power failure, such as WSP [39] and i-NVMM [5]. By using residual energy and by relying on a battery supply, WSP and i-NVMM can persist data and encryption to NVMM. However, in our case, we do not require any external battery back up for counter persistence rather rely on ECC-based sanity-check for counters recovery and application-level persistence of data.

VI. Conclusion

We propose a novel scheme called Osiris that persists encryption counters to NVM similar to strict counter persistence schemes but with significant reduction in performance overhead and NVM writes. We also present an optimized version, Osiris Plus, that improves the performance by eliminating premature counter evictions. We have shown how Osiris works and can be integrated with state-of-the-art data and counter integrity verification (ECC and Merkle tree) for rapid failure/counter recovery. We compare Osiris with other memory persistency schemes and show trade-offs in terms of hardware complexity and performance.

Our evaluation of eight representative benchmarks shows desirable performance overhead and NVM writes when employing Osiris/Osiris-Plus in comparison with other persistency schemes. For future work, we will study different ways to persist intermediate nodes in a Merkle Tree for faster recovery.

Acknowledgment

We would like to thank Dr. Zakhia Abichar and Dr. Simon Hammond for their feedback and discussion.

REFERENCES

[1] 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," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2010.

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

[3] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2009.

[4] 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: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872377

[5] S. Chhabra and Y. Solihin, "i-nvmm: A secure non-volatile main memory system with incremental encryption," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000086

[6] V. Young, P. J. Nair, and M. K. Qureshi, "Deuce: Write-efficient encryption for non-volatile memories," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694387

[7] V. Costan and S. Devadas, "Intel sgx explained." *IACR Cryptology ePrint Archive*, vol. 2016, 2016.

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

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

[10] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2. IEEE Computer Society, 2006.

[11] J. Chang and A. Sainio, "Nvdimm-n cookbook: A soup-to-nuts primer on using nvdimm-ns to improve your storage performance."

[12] A. M. Rudoff, "Deprecating the pcommit instruction," 2016. [Online]. Available: https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction

[13] 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)*, Feb 2018.

[14] 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 *2007 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2007.

[15] P. Zuo and Y. Hua, "Secpm: a secure and persistent memory system for non-volatile memory," in *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. Boston, MA: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/hotstorage18/presentation/zuo

[16] M.-Y. Hsiao, "A class of optimal minimum odd-weight-column sec-ded codes," *IBM Journal of Research and Development*, vol. 14, 1970.

[17] S. Cha and H. Yoon, "Efficient implementation of single error correction and double error detection code with check bit pre-computation for memories," *JSTS: Journal of Semiconductor Technology and Science*, vol. 12, 2012.

[18] R. Naseer and J. Draper, "Dec ecc design to improve memory reliability in sub-100nm technologies," in *15th IEEE International Conference on Electronics, Circuits and Systems, 2008. ICECS 2008.* IEEE, 2008.

[19] R. Naseer and J. Draper, "Parallel double error correcting code design to mitigate multi-bit upsets in srams," in *34th European Solid-State Circuits Conference, 2008. ESSCIRC 2008.* IEEE, 2008.

[20] T. N. Miller, R. Thomas, J. Dinan, B. Adcock, and R. Teodorescu, "Parichute: Generalized turbocode-based error correction for near-threshold caches," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2010.

[21] T. Instruments, "Discriminating Between Soft Errors and Hard Errors in RAM." [Online]. Available: http://www.ti.com/lit/wp/spna109/spna109.pdf

[22] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson, "Onyx: A protoype phase change memory storage array," in *Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems*, ser. HotStorage'11. Berkeley, CA, USA: USENIX Association, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2002218.2002220

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

[24] 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, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[25] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1186736.1186737

[26] M. Heroux, R. Neely, and S. Swaminarayan, "Asc co-design proxy app strategy." [Online]. Available: http://www.lanl.gov/projects/codesign/proxy-apps/assets/docs/proxyapps-strategy.pdf

[27] S. Swami, J. Rakshit, and K. Mohanram, "Secret: Smartly encrypted energy efficient non-volatile memories," in *Proceedings of the 53rd ACM/EDAC/IEEE Design Automation Conference (DAC), 2016*. IEEE, 2016.

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

[29] S. Swami and K. Mohanram, "Covert: counter overflow reduction for efficient encryption of non-volatile memories," in *Proceedings of the Conference on Design, Automation & Test in Europe*. European Design and Automation Association, 2017.

[30] X. Zhang, C. Zhang, G. Sun, J. Di, and T. Zhang, "An efficient run-time encryption scheme for non-volatile main memory," in *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '13. Piscataway, NJ, USA: IEEE Press, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2555729.2555753

[31] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories," *ACM Sigplan Notices*, vol. 46, 2011.

[32] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "Thynvm: Enabling software-transparent crash consistency in persistent memory systems," in *2015 48th International Symposium on Microarchitecture (MICRO)*. IEEE, 2015.

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

[34] 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 International Symposium on Microarchitecture (MICRO)*. IEEE, 2013.

[35] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3. IEEE Press, 2014.

[36] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated persist ordering," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016.

[37] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017.

[38] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual." [Online]. Available: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf

[39] D. Narayanan and O. Hodson, "Whole-system persistence," *ACM SIGARCH Computer Architecture News*, vol. 40, 2012.