Towards Low-Cost Mechanisms to Enable Restoration of Encrypted Non-Volatile Memories

Mao Ye, *Student Member, IEEE,* Kazi Abu Zubair, *Student Member, IEEE,* Aziz Mohaisen, *Senior Member, IEEE,* Amro Awad, *Member, IEEE*

Abstract—Since Non-Volatile Memories (NVMs) started entering the mainstream memory/storage market, we must consider how to secure NVM-equipped computing systems. Recent Meltdown and Spectre attacks are a strong evidence that security must be intrinsic to computing systems instead of being 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. Since different counter-mode encryption schemes are used in industry and research, we provide a versatile Osiris implementation that improves the performance and write-endurance in different memory encryption schemes. 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, and three computation-intensive graph algorithms from CRONO. Compared to a write-through counter-cache scheme, on average, Osiris can reduce 45.8% of the memory writes (increase lifetime by 1.86x), and reduce the performance overhead from 44.7%(for write-through) to only 4.49%. Furthermore, without the need for backup battery or extra power-supply hold-up time, Osiris performs better than a battery-backed write-back (4.4% vs. 5.7% overhead) and has less write-traffic (1.8% vs. 5.4% overhead).

Index Terms—Secure architectures, persistent memory, non-volatile memories, crash consistency, secure recovery.

1 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 in power loss events, NVMs represent a perfect building block for architectures that allow files to be stored in media and accessed in a way 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], giving more options for running workloads with large memory footprints.

While NVMs are certainly promising, they face 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], [5], [6]. To address such vulnerabilities, processor vendors, such as Intel and AMD, have started supporting 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 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], [7], [8], [9]. For counter-mode encryption, encryption counters/initialization vectors (IVs) are needed and are typically organized or 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 64B block granularity, e.g., DDR4 8n-prefetch mode. Moreover, packing up multiple counters in a single cache line can exploit spatial locality to increase counter cache hit rate.

Most prior research work has assumed systems have a 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, reasonably long-lasting Uninterruptible Power-Supplies (UPS) are expensive and occupy large areas in practice. 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*

- *M.Ye, K.A.Zubair and A.Awad are with the Department of Electrical and Computer Engineering, A.Mohaisen is with the Department of Computer Science, University of Central Florida, Orlando, FL, 32826. E-mail: mye@knights.ucf.edu*

Manuscript received April 19, 2005; revised August 26, 2015.

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 in 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 backup battery or expensive power-supplies is in-feasible, 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 that have not been persisted on memory during crash, can result in meaningless data after decryption. Even worse, losing the most-recent counters invites a variety of attacks that exploit reuse of 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, although the selective counter persistent scheme can be safely integrated with the monolithic counter scheme, it can cause encryption pad reuse in non-persistent data memory locations for other counter schemes. 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 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]. Moreover, our solution chooses to provide consistency guarantees for all memory blocks rather than limiting 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 when 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. 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 from SPEC2006 and U.S. Department of Energy (DoE) proxy applications, as well as three computation-intensive graph algorithms from CRONO. Compared to the write-through counter-cache scheme, on average, Osiris can reduce 45.8% of the memory writes (increase lifetime by 1.86x), and reduce the performance overhead from 44.7% (for write-through) to only 4.49%. Furthermore, without the need for backup battery or extra power-supply hold-up time, Osiris performs better than a battery-backed Write-Back (4.4% vs. 5.7% overhead) and has less write-traffic (1.7% vs. 5.4% 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. Both schemes are for split counter and local counter mode encryption.
- We propose *Osiris-global*, a counter persistence scheme that guarantee crash consistency for global counter mode encryption mode at a cost of a small overhead and number of NVM writes compared to write-through scheme, without the need for battery backup.
- We discuss several design options for Osiris and Osiris-global and provide trade-offs between hardware complexity and performance.
- We discuss the recovery advantages of our schemes, and 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 2, we briefly discuss the main concepts and background relevant to our paper. In Section 3, we discuss our own solution, its design options, trade-offs and security impact. Section 5 covers our evaluation methods and analysis for Osiris. Later, in Section 6, we discuss prior and related work. Finally, we conclude our work in Section 7

2 Background and Motivation

2.1 Background

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

2.1.1 Emerging NVMs

Emerging NVM technologies, such as Phase-Change Memory (PCM) and Memristor, are promising candidates as 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, while enabling 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.

2.1.2 Memory Encryption and Data Integrity Verification

There are several 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 the 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 the *counter mode* encryption. In the counter mode, 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 counter-mode encryption, various counter organizations are used and have distinguished structures, as described in Table 2.1.2. In state-of-the-art designs [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).

In contrast, recent work leverage local counter and global (monolithic) counter [8], [13], [14]. In such schemes, a global(monolithic) counter can be thought of as a timer, incremented on each memory write, and due to its mono-increasing property can be used to generate the OTP of each block to be written to memory. The counter value will be saved in memory along with the written block and used later for decryption. Whereas a local counter is incremented only when its associated block is written-back. During encryption and decryption, the local counter value is appended to local data address to make the IV unique. Clearly, such implementations are simple to implement, however, require high storage overhead; 64-bit counter value for each data block.

TABLE 1
Counter Types Used in Encryption Mode

Type	Structure	Incremental Policy	Overflow
Global (Monolithic)	Only one global counter	For every write-back to a data block, the global counter adds one	Frequent
Local	Each data block has one local counter	For every write-back to a data block, this block's local counter adds one.	Not frequent
Split	Each data block has a local major counter(per page) and a local minor counter(per block)	For every write-back to a block, its minor counter adds one. Every minor counter overflows, major counter adds one and all the data blocks in that page have to re-encrypt using new major and minor counter combination.	Not frequent

Similar to prior work [4], [6], [10], [13], [15], 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 bus snooping attack, dictionary-based attacks, known-plaintext attacks and replay attacks. In state-of-the-art work [10], 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 data blocks' counters in a 64B block; 64 7-bit minor counters and a 64-bit major counter. 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 memory re-encryption rate and minimizes the storage overhead of encryption counters when compared to other schemes such as global counter scheme or local counter for each cache block [13], [14]. Additionally, a split-counter scheme allows for better exploitation of spatial locality of encryption counters, achieving a higher counter cache hit rate. Although other type of counters have large storage overhead, 64-bit for each 64-byte block, easy to overflow and show high miss rate, they are still used for their unified and simple nature to implement [13], [14]. Therefore, in our work, all the counter types are discussed.

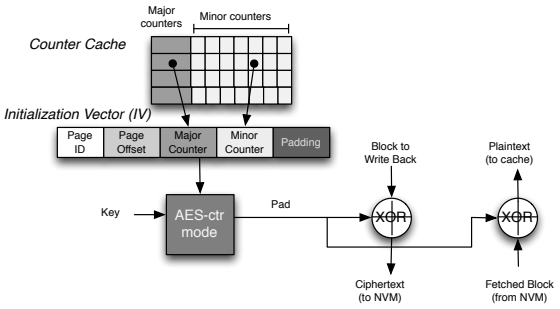

Fig. 1. State-of-the-art counter mode encryption. AES is shown but other cryptographic algorithms are possible [4]

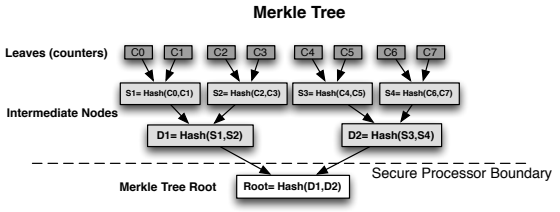

Fig. 2. An example Merkle Tree for integrity verification.

Data integrity is typically verified through a Merkle Tree — a tree of hash values with the root maintained in a secure region. In addition to the data, integrity for the encryption counters is also an essential part of the overall integrity of the system. In order to provide sufficient protection to the entire memory using a tree of reasonable size, state-of-the-art designs combine both data integrity and encryption counter integrity through a single Merkle Tree (Bonsai Merkle Tree [15]). 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.

2.2 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.

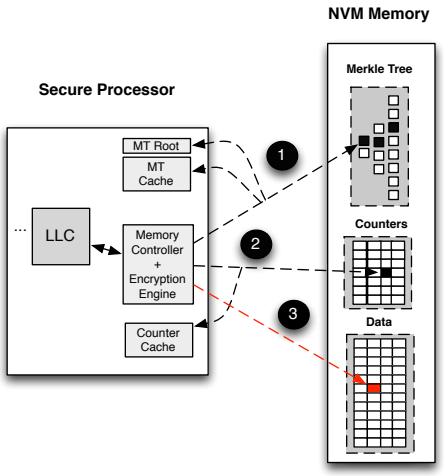

Fig. 3. Steps for write operations to ensure crash consistency.

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 in cache 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 (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 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 naturally 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 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 enough to write them to the slower NVM registers inside processor. Ensuring such 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.

2.3 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% [15]), our focus in this paper is to reduce the overhead of persisting encryption counters. To persist the encryption counters, we employ two methods as our baselines: *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 [16]. 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 [13]. 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.

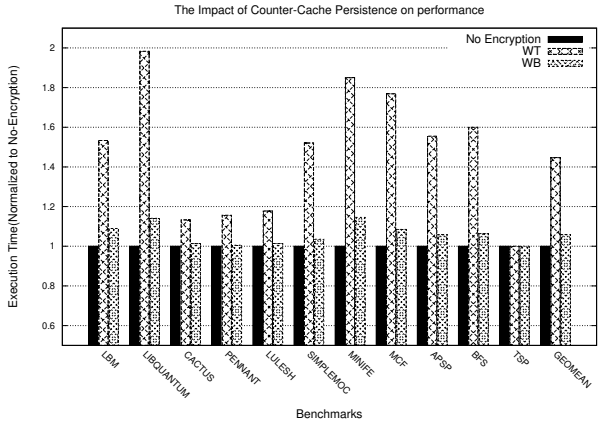

Fig. 4. Impact of counter-cache persistence on performance.

Shown in Figure 4, WT counter-cache entails a significant performance degradation for most of the benchmarks compared to no-encryption and WB counter-cache schemes. On average, WT scheme degrades the performance by 44.7% whereas WB scheme only degrades performance by 4.5% 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.

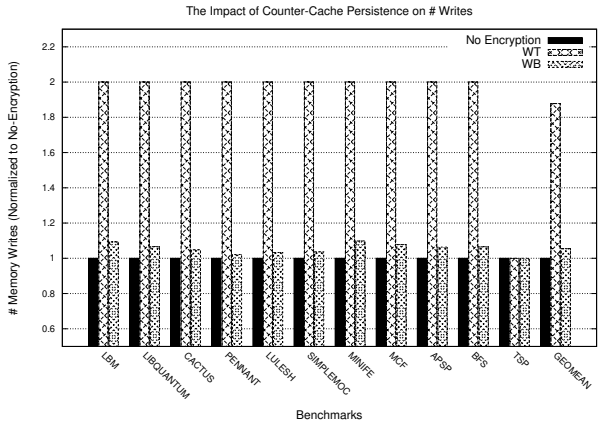

Fig. 5. Impact of counter-cache persistence on number of writes.

Another key metric relevant to NVM is the write traffic. NVMs have a limited write bandwidth due to the limited write-drivers and the large power-consumption for each write operation. More critically, NVMs have a limited endurance for writes. In WT scheme, and 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 5.4% writes on average when compared to the no-encryption scheme, whereas WT, as expected, incurs twice the write traffic.

Note that *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.

3 Osiris

We first discuss our threat model, followed by the design of Osiris and the possible design options and trade-offs.

3.1 Threat Model

Our assumed threat model is similar to the state-of-the-art work on secure processors [4], [6], [8], [13]. The trust base includes the processor and all its internal structures. Our threat model assumes that an attacker can snoop the memory bus, scan the memory content, try to tamper with the memory content (including rowhammer) and replay old packets. Differential power and electromagnetic inference attacks, as well as 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, it introduces serious security vulnerabilities. Specifically, assume an attacker application uses known-plaintext and write it to memory, however, if the memory location is non-persistent the encrypted data will be written to memory but probably not the counter. Thus, by observing the memory bus, the attacker can find out 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 $E_{key}(IV_{new})$. After a crash, however, the memory controller will read IV_{old} and increment it, which generates IV_{new} and then encrypt the new application data written to that location to become $E_{key}(IV_{new}) \oplus Plaintext2$. Knowing $Plaintext2$ only needs 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, hence 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 having a physical attacker or bus snooper.

3.2 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 happens. Without a mechanism to persist encryption counters, once a crash occurs, we 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, most likely, 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, and 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 Obtaining the corresponding most-recent encryption counter from memory. ② Verifying the integrity of data through MAC and the integrity of the used counter value through Merkle Tree. As it is possible that Step ① results in a stale counter, Step ② will fail due to Merkle Tree mismatch. Remember that the root of the Merkle Tree has been updated before crash, thus using any 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 due to 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 very 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 almost impractical if there are multiple stale counters. Let's say counters of blocks X and Y are lost, then we need to reconstruct Merkle Tree with all possible combinations/values of X and Y, and then compare the resulting root with the one safely stored in processor. While for simplicity we only mention losing two counters, in actual crash where a counter cache is hundreds of kilobytes, we will likely have thousands of stale blocks.

Observation 1. Losing encryption counter values renders reconstructing Merkle Tree nearly impossible. Approaches such as brute-force trial of possibly lost counter values to reconstruct Merkle Tree will likely take prohibitive time especially when multiple counter values have been lost. Hence, verifying the integrity/correctness of the counters stored in NVM is challenging.

One possible way to reduce reconstruction time is by employing ***stop-loss*** mechanisms to 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 searching 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, stop-loss mechanism could reduce the time to reconstruct the Merkle Tree, however, still is 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 was the most recent counter value 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 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 the 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 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 under utilization 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 the 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.

3.3 Design

Osiris mainly relies on inferring the correctness of an encryption counter from calculating the ECC associated with the decrypted text and compares it with that encrypted and stored along with the encrypted cache line. 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 along with the cache line. In other words, the tuple 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 encrypt it with the cache line before writing both to memory. ② The second option is to encrypt the plaintext, then calculate the ECC over the encrypted block before both are written to the memory. Although approach ② allows overlapping decryption and ECC verification, most ECC implementations used in memory controllers, e.g., SEC-DED Hsiao Code [17], take less than a nanosecond to complete [18], [19], [20], which is negligible compared to cache or memory access latencies [21]. 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 the counter-mode encryption, the data is decrypted using the following: $\{X, Z\} = E_{key}(IV_X) \oplus Y$, where Y is 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 the counter-mode encryption is used, the reason could be an error (or tampering) occurred on X or Z, or *wrong IV* is used to do the encryption, i.e., decryption is not successful.

Observation 3. When the ECC function is applied over the plaintext and the resulting ECC bits are encrypted along with the data, ECC bits can provide a sanity-check for the used encryption counter. Any tampering with the counter value will be detected by a *clear* mismatch of the ECC bits result from that invalid (wrong/stale counter) decryption; results of $E_{key}(IV_{old})$ and $E_{key}(IV_{new})$ are very different and independent. Note that in 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 used counter can be used, the ECC bits can fail to provide guarantees as strong as cryptographic MAC values. Accordingly, we adopt Bonsai Merkle to additionally protect 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 the ith encrypted data bit 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 results 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 uses CTR-mode for security and performance reasons.

The question is how to proceed when there is an error detected due to a mismatch between the expected and stored ECC. As the reader would expect, the first step is to find if the error is correctable using the ECC code. If the error is uncorrectable, before giving up, we take the odds that the IV used is incorrect, i.e., the decryption was not successful. Our goal is to find out if the error is due to using a wrong IV. Below is a summary of the common reasons for such a mismatch between the expected ECC and the stored ECC after decryption:

TABLE 2
Common Sources of ECC Mismatch.

Error Type	Typical 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 it**

As shown in Table 2, one reason for such a mismatch is using an old IV value. To better understand how this can happen, we recall the counter cache persistence issue. If a cache block is updated in memory, it is necessary to also 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 such an IV was actually the one used to encrypt the block, but 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 using a relaxed counter persistence scheme employing 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 one used to encrypt the block. Thus, Osiris deploys the stop-loss mechanism to limit such possibility to only N counter updates; 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 the baseline, all counters being read from memory reflect their most-recent values during normal operation, and the most-recent value is either updated in cache or 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 write-back mechanism by updating memory counters 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 case of miss. Accordingly, Osiris operation in normal operation is similar to conventional memory encryption except that a counter is additionally persisted once each Nth update, hence acting like a write-through for the Nth update of each counter.

In contrast, Osiris-Plus allows occasional dropping of most-recent values of encryption counters through relying on run-time recovery mechanism of the most-recent values of counters. In simple words, Osiris-Plus relies on trying multiple possible values on each counter miss to recover the most-recent one before verifying it through Merkle Tree, whereas baseline Osiris would do that only at system recovery time.

System Recovery Stage: During system recovery time, both Osiris and Osiris-Plus need to reconstruct Merkle Tree at the time of restoration. Furthermore, both need to use most-recent values of counters to reconstruct a Merkle Tree that eventually has a root that matches the root stored in the secure processor. In both Osiris and Osiris-Plus, the system recovery will start with traversing all memory locations (cache lines) in integrity verified region (all memory in secure NVM). For each cache line location, i.e., 64B address, the memory controller uses 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 will 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 vetted, 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.

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

3.3.1 Osiris Read and Write Operations

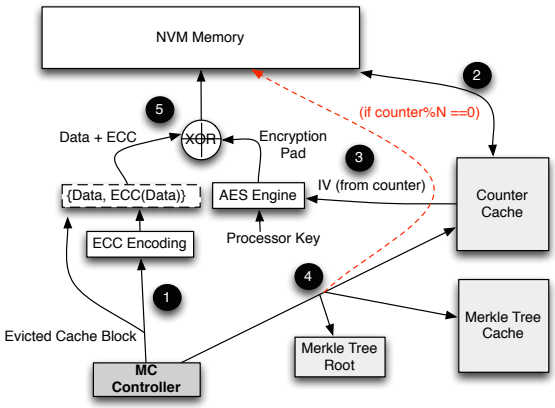

Fig. 6. Osiris write operation

As shown in Figure 6, during a write operation, once there is a cache block 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 typically are buffered for a while in the write-pending queue. Later, in Step ②, the memory controller obtains the corresponding counter in case of 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. Later, in Step ④, the obtained counter value will be verified (in case of miss) and then the counter and affected Merkle Tree (including root) are updated in Merkle Tree cache. Additionally, unlike typical write-back counter-cache, if the new counter value is a multiple of N or 0, then the 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.

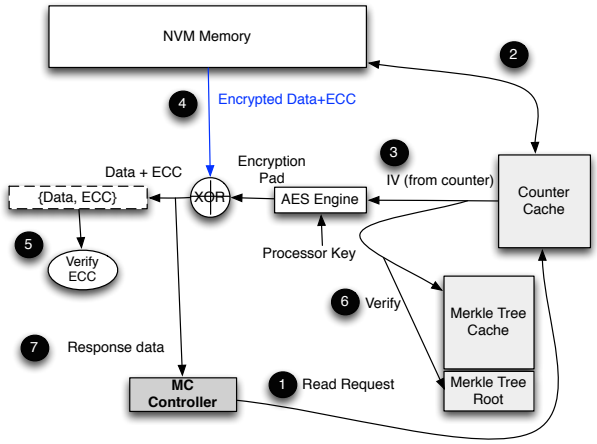

Fig. 7. Osiris read operation

During read operation, as in Figure 7, the memory controller will obtain the corresponding counter value from counter cache (with hit) or memory (with miss) and evict the victim block, if dirty, as shown in Steps ① and ②. Later, the obtained 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. Later, in Step ⑤, traditional ECC checking occurs to the decrypted data. Finally, before proceeding, 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 with any conflicts, for instance, Step ⑥ and steps ④ and ⑤.

3.3.2 Osiris-Plus Read and Write Operations

The write operation in Osiris-Plus is very similar to the write-operation in baseline Osiris except that it does not write back evicted dirty blocks from counter cache (as could happen in Step ② of Figure 6); Osiris-Plus recovers the most-recent value of 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.

The main difference between Osiris and Osiris-Plus are 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

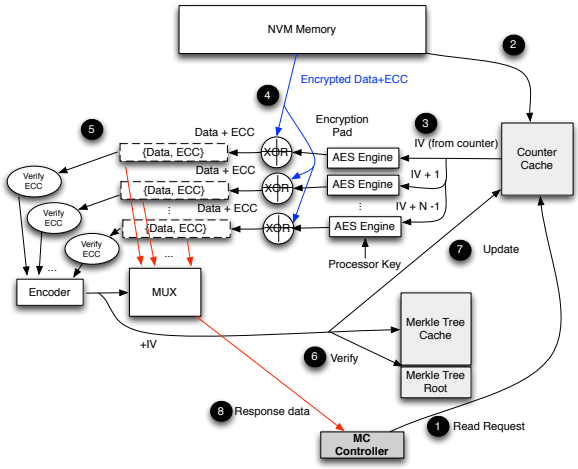

Fig. 8. Osiris-Plus read operation

extra cycles. Later, once a counter value is detected, through post-decryption ECC verification, to be the most-recent one, it will be verified through Merkle Tree similar to baseline Osiris. Later, once 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 with the recovered value, as shown in Step ⑦.

3.4 Hardware Overhead

The major hardware components added for our design are the parallel AES engines. Since the commercial chips with AES engines, i.e chips made for Apple iOS, NXP® C29X, MAX36025, do not reveal the statistics about the area and power consumption of their implemented AES engines [22], [23], [24],we estimate the power consumption and area cost based on the statistics from [9]. For each 128-bit AES engine, the power cost is $15.1mW$ and the area cost is $0.204mm^2$. So for 4 parallel AES engines, it incurs about $60.4mW$ energy overhead and requires $0.816mm^2$. Compared to the total power usage $70W$ [25](ranging from $40W$ to $85W$) and multiprocessor area size $246mm^2$ [26] ($102mm^2$ to $768mm^2$ since 2012) of a chip, our implementation will introduce the overhead of less than 0.1% of power usage and around 0.33% of area on die.

3.5 Reliability and Recovery from Crash

As mentioned earlier, to recover from crash, Osiris and Osiris-Plus need to first recover the most-recent counter values through utilizing post-decryption ECC bits to find the correct counters. Later, the Merkle Tree will be constructed and the root will be verified and compared with that kept in the root. While the process seems reasonably simple without errors, the recovery process can get complicated in the presence of errors. Specifically, uncorrectable errors in the data or encryption counters render generating a matching Merkle Tree root nearly impossible, and hence the inability 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 Merkle Tree can potentially fail the whole recovery process. Specifically, when the system attempts to reconstruct the Merkle Tree, it will fail to generate a matching root. Furthermore, it is also infeasible to know which part of the Merkle Tree causes the problem; only the root is maintained and all other parts of Merkle Tree should work perfectly and generate the same root or none is trusted. One way to mitigate such single-point of failure is to keep other parts of the Merkle Tree in the processor and guarantee they are never lost from the secure processor chip. For instance, for an 8-ary Merkle Tree, the immediate 8 children of the root are also saved all the time in the processor. In a more capable system, the root, its immediate 8 children and their immediate children are kept, which is a total of 73 MAC values. In case recovery fails to produce the root of the Merkle Tree, we can look at which children are mismatching, and then declare that part of the tree as un-verifiable and probably warn the user/OS. While we provide such insights to solve 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 Merkle Tree can be implemented in case of high error systems. We leave reconstructing Merkle Tree in 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 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 [27], 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 out 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%. Even $P_{k\geq 7}$ is 99.95%. Note that the probability all 8 codewords indicate an error is 96.91%, which is still very high. In other words, Osiris can successfully identify wrong counters with a success rate of almost 100% by filtering out any counter has 7 or more ECC codewords indicating errors. Thus, except for the pathological case where a cache block

has actual errors on each of its words, Osiris can reliably distinguish wrong counters with a success rate of almost 100%. In that extremely pathological case, where each word of the memory block has at least one error, all counter values will be filtered out, and then Osiris can verify all the counters values (e.g., 8 values) through Merkle Tree to recover the correct one. Note that the probability a bit error occurs is very small, however, the described pathological case occurrence requires at least an error to occur on each of the 8 memory words in the block at the same time, which is extremely rare. Note that the words and ECC are typically interleaved and spread in the row and thus nearly independent [27]. It is also important to note that all of our discussion here about detecting a wrong encryption counter in the presence of errors is relevant to the case of having an error.

In summary, Osiris can detect wrong counter reliably in all cases except the pathological case where there is at least an error *on each word* of the block corresponds to the counter being recovered, which will require an additional step of verification through Merkle Tree. Thus, Osiris does not affect how many errors an ECC can detect/correct per word but limits the number of faulty words within a 64B to not exceed 7 words to avoid additional step (Merkle Tree) before detection/correction. We believe that having errors on each word of a block is an extreme case and will not affect the adoption of Osiris. Note that some 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 [28], which even makes our detection success rate even closer to perfect.

3.6 Systems without ECC

Some systems do not employ ECC bits, but rather rely on MAC values that can be co-located with data instead of ECC bits [8]. For instance, the ECC chips in the DIMM can be utilized to store the MAC values of the Bonsai Merkle Tree, and hence allow obtaining data integrity-verification MACs along with the data. While our description of Osiris and Osiris-Plus was focused on using ECC, MAC values can also be used to achieve the exact same purpose; sanity-check for the used decryption counter. The only difference is that if there is an error, when ECC bits are used, we can use the number of mismatching ECC bits as a way to guess the counter value, whereas MAC values tend to differ significantly when there is any error in data. Accordingly, to mitigate 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, can be used, thus 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. Note our proposed schemes also work with ECC and MAC co-designs such as Synergy (parity+MAC) [8].

3.7 Security of Osiris and Osiris-Plus

Since Osiris and Osiris-Plus rely on final verification step through Merkle Tree, Osiris and Osiris-Plus have security guarantees and tamper-resistance similar to any scheme that uses Bonsai Merkle-Tree, such as state-of-the-art secure memory encryption schemes [4], [6], [8], [13].

4 Osiris-Global

As mentioned earlier, encryption counters can be organized in different ways; global, local or split counters (Table 2.1.2). The global (monolithic) counter referred here is similar to the central counter used in SGX enclave [14], which is incremented by one for every writes to the main memory. On the other hand, local counters are per-block counters that are incremented whenever associated memory block is written. Finally, in split counter scheme, subsequent encryption of the same block would just increment its split counter value (7-bit minor counter) by one.

As discussed previously in section 3.3, *Osiris* employs stop-loss mechanism in consecutive increment of per-block counters. However it cannot be directly applied to global/monolithic counter scheme where only one global counter records all the writes to the data blocks. In global/monolithic counter scheme, each write will increment the global counter, and the corresponding written block will copy the value of the global counter as its own counter value. Hence the subsequent updates to a specific block might lead to a large increment of the associated global counter value. This is due to the fact that, many encrypted writes to other blocks are possibly performed in between these two subsequent writes. Consequently, employing *Osiris* in monolithic counter scheme is challenging as the counter value for a specific block does not increment consecutively between consecutive block updates, upon which *Osiris*'s recovery is based. It is impractical to try out all the possible values less than or equivalent to the global counter value to match the Merkle Tree root, since this immense number of trials could be multiplied by thousands of lost blocks (due to crash).

Even though Osiris can not be directly applied to global counter scheme, we still can use a method that reflects Osiris's spirit to solve this problem. Similar to phase number N in Osiris, we will take advantage of a much larger *epoch number* (EN), compared to the N value used in Osiris. The goal of this design is to guarantee that for each stale counter in memory we can find its up-to-date counter value within an interval of [Global Counter Value - EN, Global Counter Value]. To achieve that, we want to persist counters in cache after every EN writes.

In a naive implementation, whenever a result of the global counter modulo EN is equal to zero, all the cache lines in the counter cache could be persisted in main memory. As such, if a crash happens, the lost counter values in the counter cache will be at most EN away from those that have been persisted in main memory. To recover the lost counters, we can again use ECC bits to identify correct counters from all possible counter values as depicted in *Osiris* scheme (Section 3). However, this process definitely demands more recovery time compared to *Osiris* implementation, since the value of EN is significantly larger than the phase number N used in general *Osiris* scheme. Apart from demanding longer recovery time, this naive implementation can also slow down or stall the system due to the frequent flushing of entire cache. Note that EN value is negatively correlated with the frequency to persist cache lines to main memory. If the value of EN is chosen to be small, recovery time could be reduced at the cost of higher frequency to persist cached

counters. Therefore, the EN value must be chosen carefully. We expect it to be a relatively large value, in hundreds, or thousands to reduce the persistence overhead.

To ensure low overheads, we leverage a scheme that endeavors to reduce the redundant persistence of counters as much as possible and guarantees the lost counters will be within the range of [Global Counter-EN, Global Counter]. To this end, we require a volatile hardware-based Epoch Reference Table (ERT) (Figure. 9) with EN entries, i.e., 1024 as shown in the Figure 9, and it has 8KB for 8 byte counter addresses and 8KB for 8 byte timestamp, to track the history of the past EN writes. Additionally, we need to add a column in counter cache: the timestamp (64-bit) (Figure. 9). Epoch Reference Table serves to record the counter value address and the counter updated time for every memory write. On each write in the counter cache, we consult the table to find out which counter address was written EN writes ago by checking the entry index of (Global Counter Value % Epoch Number), and check whether the counter value of that old address needs persistence in NVM. If yes, we persist the counter value to main memory and update the timestamp of the corresponding cacheline in cache. Finally, we need to update the entry of current index with the latest counter write address and write/persistent timestamp.

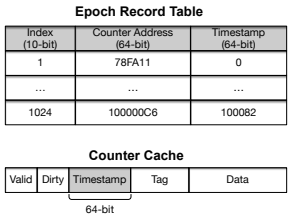

Fig. 9. The structure of the Epoch Record Table (ERT) and the counter cache

The key idea here is that, to determine whether to persist a counter relies on the comparison result of timestamps for that counter both in the ERT and in the counter cache. It should be noted that, if the checked counter cannot be found in the cache, or it is a hit without dirty-bit set, it must have been evicted to the memory within last EN writes. Hence no further efforts need to be done in these cases.

The timestamps in the cache either indicates the latest update time or the persistence time to main memory. The timestamp in Epoch Reference Table always reflects the timestamp the new entry is recorded. Note that, for all the newly inserted cache lines, their timestamps in cache are set to 0. Thus if a timestamp of an old entry in the ERT is larger than the timestamp of the cache line (assuming that it is found in the cache) in condition that cache line has the dirty bit set, it indicates the cache line has not been persisted within recent EN updates. Therefore, we have to persist it before evicting the old entry. Similarly, if a checked cache line has a later(larger) timestamp than that from a table entry, it means the counter in cache has been persisted recently and hence needs no redundant persistence again. That is to say the checked table entry can be safely evicted for a new counter update content. The detailed scheme of Epoch Reference Table update and counter persistency is displayed in the algorithm 1. To help the readers to understand the whole picture of the algorithm, we also draw a flowchart 10 to diagram the algorithm.

Algorithm 1: Epoch Reference Table Update and Counter Persistency

Data: Counter Cache(CC), Epoch Reference Table(ERT), Epoch Number(EN), Global Counter(GC)
Result: Update ERT for each counter write, and persist counter recorded in ERT EN writes ago conditionally

```
1  Initialization;
2  Timestamp in all cachelines is set to 0;
3  Timestamp in any inserted cachelines is set to 0;
4  All colunms in ERT are set to 0;
5  GC = 0;
6  while MC gets a write request do
7      Update counter in cacheline;
8      TimePoint CurrentTime = current time;
9      GC++;
10     Index i = GC%EN;
11     if ERT[i].Address in CC and Dirty then
12         Tag T=ERT[i].Address/Data blocks' size for a
            cacheline;
13         if CC[T].Timestamp ≥ ERT[i].Timestamp then
14             ERT[i].Address = Counter address with GC
                value;
15             ERT[i].Timestamp = CurrentTime;
16         else
17             Write CC[T] to main memory;
18             CurrentTime = current time;
19             CC[T].Timestamp = CurrentTime;
20             CC[T].Dirty = False;
21             ERT[i].Address = Counter address with GC
                value;
22             ERT[i].Timestamp = CurrentTime;
23         end
24     else
25         ERT[i].Address = GCth write's counter address;
26         ERT[i].Timestamp = CurrentTime;
27     end
28 end
```

When using this scheme, our table behaves like a circular buffer that helps us know which counter block was updated EN writes ago. We need flush the block if it exists in the cache and is dirty and has not been recently persisted. By doing so, we can ensure that any updated counter block, if gets lost due to a crash, has a value between $[GloablCounter - EpochNumber, GlobalCounter]$.

4.1 Write Operation of Osiris-global

The write operation for Osiris-global is shown in Figure 11. The distinguished step in write operation is shown in ③. After updating the latest counter for the coming write request in cache, we need to find the entry index in Epoch Reference Table to record this update. Before updating an entry in the ERT, we will check the old entry in cache. There are two cases in which we proceed directly to steps

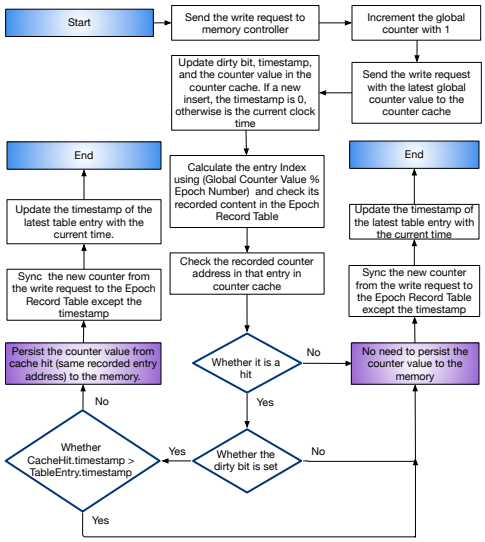

Fig. 10. The flowchart of Epoch Record Table (ERT) update and counter cache update

⑥ and ⑦ after overwriting the old counter address and timestamp in the table with the latest write information. First, if the address in an entry is present in cache, dirty, and its timestamp in the table is larger than the timestamp in cache. Second, the counter address of an entry is either not in cache or not dirty. Otherwise, we have to persist the counter in the cache to NVMM before moving on to the subsequent steps.

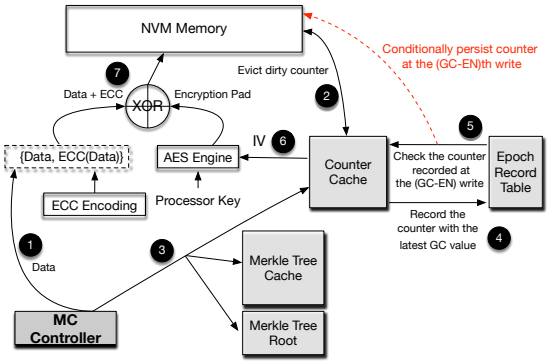

Fig. 11. Osiris-global write operation

4.2 Hardware Overhead for Osiris-global

For Osiris-global, the hardware complexity involves a hardware table(Epoch Record Table) and a timestamp column in the cache. The hardware table has a size less than 18K and the size of timestamp column of 64-bit roughly introduces less than 12.5% additional counter cache size. Both are within an acceptable overhead range.

5 Evaluation

In this section, we evaluate the performance of Osiris with other state-of-the-art persistence and recovery schemes. Section 5.1 describes the methodology we adopt for our experiments. Section 5.2 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 3
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 [29]
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], [29] tCL 12.5ns, 64-bit bus width, 1200 MHz Clock
Encryption Parameters	
Counter Cache	256KB, 16-way, 64B block

5.1 Methodology

We model Osiris in Gem5 [30] with the system configuration presented in the Table 3. We implement a 256KB, 16-way set associative counter cache, with a total number of 4K counters for Osiris and Osiris Plus scheme. Similarly,for Osiris-global, we implement a 256KB, 16-way, set associative counter cache based on our cache sensitivity studies. To stress-test our design, we select memory-intensive as well as computation-intensive applications to evaluate all the Osiris schemes. Specifically, we select eight memory-intensive representative benchmarks from the SPEC2006 suite [31] and from proxy applications provided by the U.S. Department of Energy (DoE) [32]. For computation-intensive applications, we use three graph algorithms from CRONO [33]. 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 battery while minimizing the performance and write traffic overheads.

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

Osiris-global Scheme: This is a scheme for global monolithic counter which persists the counters by checking counter's timestamp with its active counterpart in cache if present, and maintaining an epoch reference table. This scheme does not need battery support, however, due to the large *Epoch Number*, it will cause multiple-cycle overhead to identify the correct counter value for each stale counter. Meanwhile, the number of stale counters and recovery time is bounded by the *Epoch Number*

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 mode encryption scheme with a dedicated counter cache. The WB scheme only writes to memory dirty evictions from counter cache. However, WB scheme assumes a battery is sufficient to flush all dirty blocks in counter cache.

5.2 Analysis

As discussed in Section 3.3, the purpose of Osiris/Osiris-Plus is to persist the encryption counters in NVM memory in response to system crash recovery, however, with reduced performance overhead and write traffic. Therefore, the selection of number N for update interval (also discussed in Section 3.3) 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.

5.2.1 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. Note that only SPEC2006 and DOE Mini Benchmarks are used for determining Osiris limit and the average performance of WB and WT are also based on these benchmarks. The rest of the paper uses all three type benchmarks including CRONO. Figure 12 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.

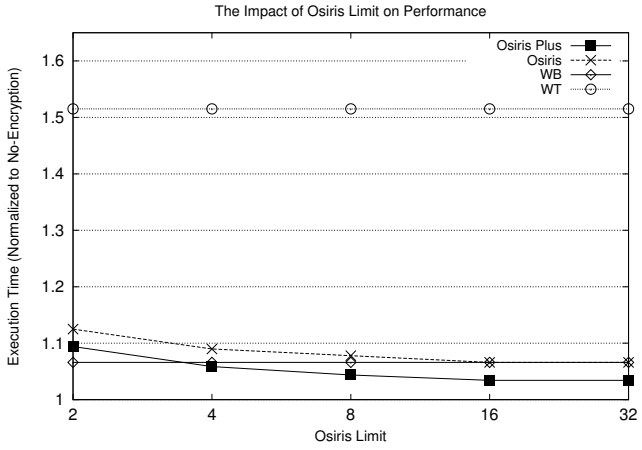

Fig. 12. The impact of Osiris limit on performance.

From Figure 12, 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 the Figure 12, 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.

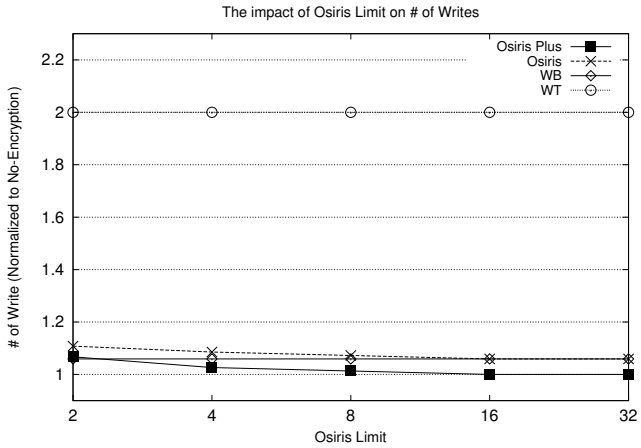

Fig. 13. The impact of Osiris limit on number of writes.

A similar pattern is observed in Figure 13 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, while Osiris has write-traffic overhead similar to WB. As such, we use limit 4 as a reasonable trade-off between the performance overhead and the required additional stages or extra checking at the time of recovery.

5.2.2 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 5.2.1.

As we can observe from Figure 14 and Figure 15, 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.

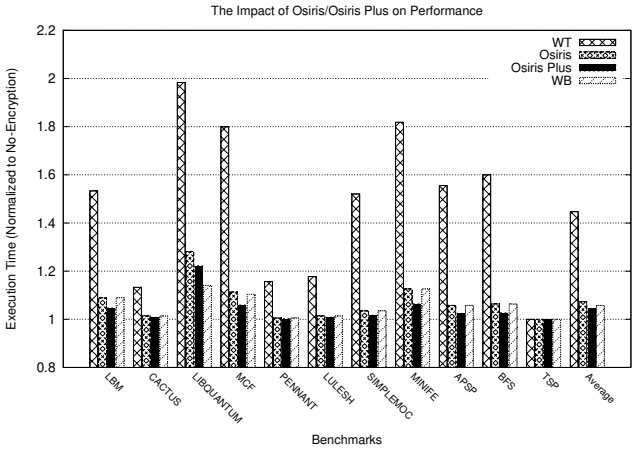

Fig. 14. Osiris and Osiris Plus' persistence impact performance.

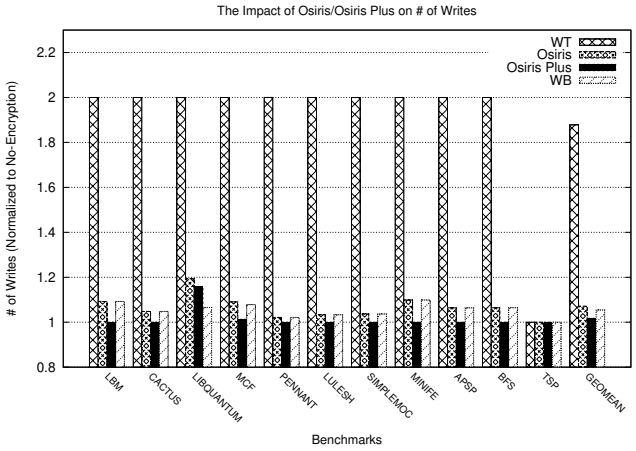

Fig. 15. Osiris and Osiris Plus' persistence impact the number of writes.

However, since Osiris Plus can additionally eliminate premature (before Nth write) counter evictions, it can actually outperform WB. There are two exceptions, however. One 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. Another exception is `TSP`(Traveller Salesman Problem), which is an NP problem that requires tremendous computation, but no need for any memory-write access(See Figure 15). Since no write is involved, any write-optimized schemes will not help this application in terms of performance, no exception for Osiris or Osiris Plus. Similarly, WT will work the same as WB scheme. That is why we see no performance gain or difference among all the schemes evaluated. 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.

5.2.3 Impact of Epoch Number

Recall that in Osiris-global, we introduce an epoch number(EN). Accordingly, before we record updated counter address present in counter cache to an entry of Epoch Reference Table, we need to check whether to persist a counter address written EN times ago in that specific entry. First, we did a sensitivity study to determine the appropriate EN to use in our experiments and the result is shown in Figure 16. The comparison demonstrates that the larger the EN is, the impact on performance is more close to WB scheme and with a longer recovery time because of too many trials one has to attempt to identify a data-and-ECC match. As such, we pick 1024 as our Epoch Number.

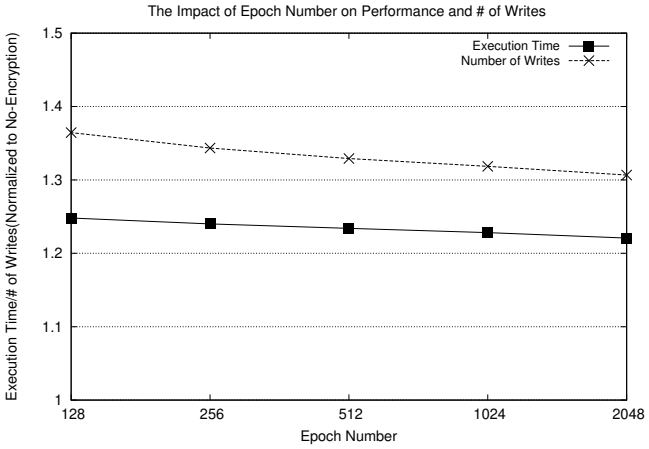

Fig. 16. Epoch Number's impact on performance (Number of writes).

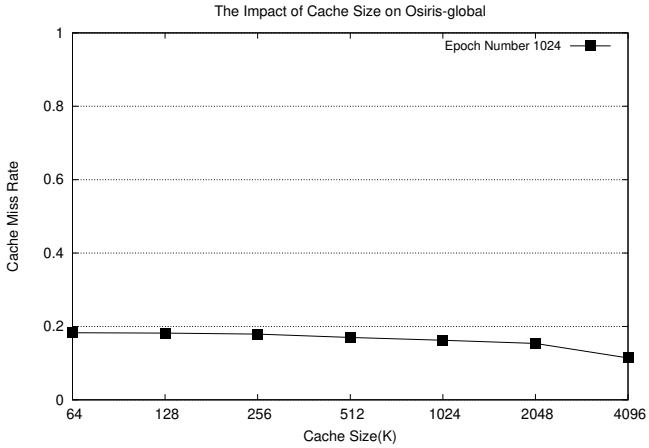

Fig. 17. The sensitivity studies of cache size for Osiris-global

5.2.4 Osiris-global persistency on multiple benchmarks

As we did for Osiris/Osiris Plus, we conducted the sensitivity study for Osiris-global. Since global counter compared to split counters does not provide good spacial coverage, hence the overall counter cache miss rate is high, even at the size of 4M(11.4% miss rate) in Figure 17. It is admitted that the miss rate decreases with a large cache size but such performance

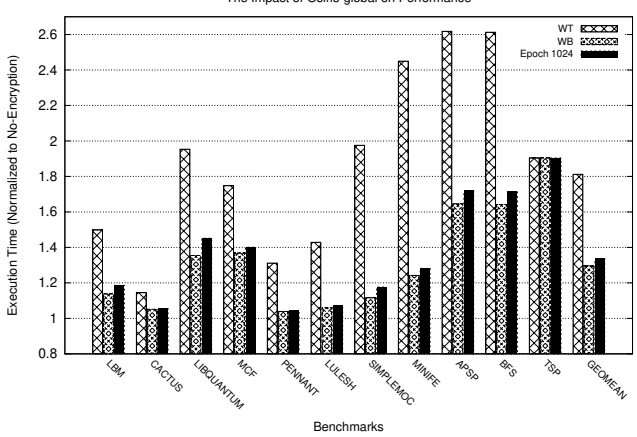

Fig. 18. The impact of Osiris-global on performance.

improvement is not very significant. When increasing the cache size from 256K to 4M, it only reduces the miss rate by 6.5% . Considering a cache takes the major space of the processor chip and requires large amount of transistors and energy, we traded off the miss rate gains for small size of 256K as the cache size for Osiris-global.

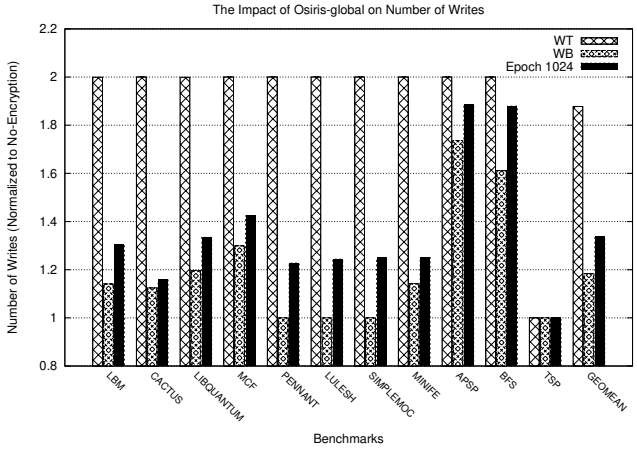

Fig. 19. The impact of Osiris-global on number of writes.

Our results in Figure 18, show that using EN of 1024, the overhead of Osiris-global on average is comparable to that of Osiris/Osiris Plus. On average, the execution time overhead is an extra 3.88% to WB scheme. For some application, such as `Libquantum`, the percentage of overhead normalized to WB scheme could reach as high as 10%, which is consistent with its performance under Osiris scheme. For computation-intensive graph algorithms, their behavior on performance are similar to the memory-intensive application except TSP and we know it is because TSP is a no-write application. Meanwhile, the average number of writes under Osiris-global scheme in Figure 19 causes additional 17.2% overhead in comparison to WB scheme and is 62.4% lower than WT scheme. Besides, `Pennant`, `Lulesh` and `Simplemoc` contribute the most write overhead, all above 20%. This phenomenon is due to temporal write correlation pattern of application update. For graph algorithms, we observe a closing gap between WT and WB in terms of number of writes. This is because graph algorithms have a higher miss rates than their memory-intensive counterparts (3-folder higher), which leads to frequent write-backs. In summary, the results obtained for performance and number of writes under Osiris-global matches our expectation, that is, to outperform WT scheme but is worse than WB scheme in an acceptable range.

5.2.5 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. When multiple AES are introduced onto a chip using Osiris Plus scheme, due to parallelism, the recovery time will be even shorter than what we estimated.

Similarly, for the monolithic counter scheme, the additional recovery time is bounded by $131072 \times 1024 \times 100ns$, which is roughly 13.1 seconds. However, note that the overhead is constant and only affected by the cache size, thus for large memory sizes the overhead would be marginal. For instance, if we use 1TB memory, the recovery time overhead will be less than 1%, and becomes much smaller for larger memories. After a second thought, since we are unlikely to persist counters within the range of [Global Counter-Epoch Number, Global Counter] after a crash event, it also means there are Epoch Number of stale counters present in memory at most if not considering other uncorrectable errors on counters. So the recovery time can be reduced to $1024 \times 1024 \times 100ns$, which is only 0.102 seconds. Although we can take use of multiple AES engines for Osiris-global for recovery, it is not necessary because Osiris-global does not affect regular read/write operation like Osiris plus does.

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.

6 Related Work

In the following, we review the related NVM security work, mostly with respect to encryption and memory persistence.

NVM Security: Most research work advocate the 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], [34], [35], [36], [37]. Since one of the fundamental works that introduced the global,local counter schemes, and devised split counter [10], optimization works are 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 [34] integrates zero-based partial writes with XOR-based energy masking to reduce both overhead and power consumption for encryption. ASSURE [35] 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] describes 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. The latest work [38] also uses de-duplication to increase the endurance of secure NVM. 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 with 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 the hardware and software research communities [39], [40], [41], [42]. Pelley [43] refers memory persistence to constraints of write order with respects to failure and proposes several persistence models in either strict or relaxed way. DPO [44] and WHISPER [45] propose persistent frameworks using strict and relaxed order, respectively, with different granularity. In our design, we assume conventional persistence model, i.e., cacheline flushing ordered by store-fencing, e.g., `CLWB` then `SFENCE`, to persist memory locations [46], and we focus on the persistence from a hardware perspective.

The most related work to ours is counter-atomicity [13]. Counter-atomicity introduces a counter write queue and a ready bit to enforce the write persistence of data and counter from the same request to NVM at the same time. To reduce the write overhead, counter-atomicity provides some APIs for users to selectively persist self-defined critical data along with their counters, hence relaxes the need for all-counter persistence. Another work that came later also tried to solve a similar problem [16]. In their work, they use write-through cache policy to guarantee the crash consistency. For transaction log and data, they use split counter spacial coverage to reduce the write number. Osiris tackles the same problem but through 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 3.1. The work discussed in this paper considers failure as a broad concept [43], whereas some work specifically handle power failure, such as WSP [47] and i-NVMM [5]. Using residual energy and by relying on 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 data persistence.

7 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 propose an optimized version, Osiris Plus, that further improves the performance by eliminating premature counter eviction. We then discuss how Osiris can work and be integrated with state-of-the-art data and counter integrity verification (ECC and Merkle tree) for rapid failure/counter recovery. Additionally, we devise an Osiris Global scheme for global(monolithic) encryption counter. The scheme is write-friendly compared to the strict WT scheme but is not as optimal as WB scheme with battery support. But it significantly reduces the recovery time for stale counters. We use our scheme to compare with other memory persistency schemes in respective of trade-off between hardware complexity and performance. Our evaluation in eleven representative benchmarks shows desirable performance overhead and NVM writes via employing Osiris/Osiris Plus in comparison with other schemes discussed. For future work, we will study different ways to additionally persist Merkle Tree intermediate nodes for faster recovery.

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] S. Gueron, "A memory encryption engine suitable for general purpose processors." https://eprint.iacr.org/2016/204, 2016.

[15] 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.

[16] 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

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

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

[19] 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.

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

[21] 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.

[22] Apple, "ios security," https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf, 2018.

[23] J. Harrison, "Crypto chip has two aes encryption modules," https://www.electronicproducts.com/Analog_Mixed_Signal_ICs/Communications_Interface/Crypto_chip_has_two_AES_encryption_modules.aspx, 2012.

[24] N. Semiconductors, "C29x: Crypto coprocessor," https://www.nxp.com/products/processors-and-microcontrollers/additional-processors-and-mcus/application-specific-mcus-mpus/c29x-family-of-crypto-coprocessors/crypto-coprocessor:C29x, 2016.

[25] W. Luk and M. J. Flynn, *Computer System Designs: System-on-Chip*. John Wiley & Sons, 2011.

[26] J. Walton, "The broadwell-e review," https://www.pcgamer.com/the-broadwell-e-review/, 2016.

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

[28] 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

[29] 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.

[30] 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," *FIGARCH Comput. Archit. News*, vol. 39, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[31] 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

[32] 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

[33] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores," in *2015 IEEE International Symposium on Workload Characterization*. IEEE, 2015.

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

[35] 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.

[36] 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.

[37] 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

[38] P. Zuo, Y. Hua, M. Zhao, W. Zhou, and Y. Guo, "Improving the performance and endurance of encrypted non-volatile main memory through deduplicating writes," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018.

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

[40] 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.

[41] 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.

[42] 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.

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

[44] 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.

[45] 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.
[46] 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
[47] D. Narayanan and O. Hodson, "Whole-system persistence," *ACM SIGARCH Computer Architecture News*, vol. 40, 2012.

Mao Ye is currently a Ph.D. student at the University of Central Florida (UCF). She got her Computer Science Master degree from UCF. Her research interests is non-volatile memory system, file system, and security. She got her bachelor degree from Zhejiang University. Before she joined the lab she worked several years in IT infrastructure department of a national bank in China.

Kazi Abu Zubair is a first-year Ph.D. student majoring in Electrical Engineering at the University of Central Florida. His research interest is in hardware security and secure computer architecture. He received his bachelor degree from the University of Chittagong, Bangladesh worked in the R&D of several startup companies in Bangladesh before joining UCF.

Aziz Mohaisen is an Associate Professor at the University of Central Florida (UCF). His research interests include computer systems security. He holds 10 U.S. patents and is an ACM and IEEE member. Currently, he leads the Security and Analytics Lab (SEAL) at UCF.

Amro Awad is an Assistant Professor at the University of Central Florida (UCF). His research interests include secure hardware architectures, non-volatile memories and hardware/software co-design. He holds several U.S. patents and he is IEEE member. Currently, he leads the Secure and Advanced Computer Architectures (SACA) research group at UCF.