Recent Security Techniques in (Non-Volatile) Main Memories

Ardhi Wiratama Baskara Yudha*[1] and Maryam Babaie †[1]

[1]Department of Computer Science, University of Central Florida

Abstract

As Non-Volatile Memories (NVMs) are becoming more and more popular and promising in the market, more attention have been attracted on their security. As they are able to retain data even after power loss, they are vulnerable against physical attacks such as stolen main memory or bus snooping. Thus, it is important to make them secure using security techniques. However, due to their feature such as limited write endurance and long write latency, security mechanism must be as efficient as possible. Moreover, such methods must be recoverable over power cycles, to ensure data recoverability of NVMs. We will briefly survey through five of such techniques that each of them try to achieve a better performance or recoverability for NVMs.

Index terms— NVM, Counter Mode Encryption, Security

1. Introduction

Emerging Non-Volatile Memories (NVMs) are a promising advancement in memory and storage systems. They are allowing memory and storage to get unified. NVMs promise large capacities and much better scalability compared to DRAM, while maintaining data persistently and closer to the processor. Secure memory systems usually use counter-mode encryption. In counter-mode encryption, each cacheline is corresponded to a counter that is used along with processor key to encrypt and decrypt the data block when it is accessed in memory. Eventhough confidentiality of counters is considered unnecessary, their integrity must be ensured because replaying an old counter would compromise the security of counter-mode encryption. For this purpose, Merkle Tree is usually used. Merkle Tree consists of hashes over hashes where the leaves are the counters, and finally the root of the tree is always kept on chip. Each counter update changes such root value, and hence any tampering will be detected due to root mismatch.

In this paper, we will review five of the most recent techniques published in the past two years in security and performance improvement of the non-volatile main memory. The techniques include: DeWrite [1], Morphable Counters [2], Osiris [3], Anubis [4], and Triad-NVM [5]. We need to mention that Morphable Counter technique is not exclusively for NVM and can be applied to DRAM.

*yudha@knights.ucf.edu
†babaie@knights.ucf.edu

2. DeWrite

2.1. Problem Description

First, NVM has limited Write endurance. NVM has higher latency and energy overhead for write than read. In NVM as persistent memory, for data consistency it must ensure the ordering of memory writes which makes the write on the critical path of execution. Thus, reducing write operations is vital in NVMs.[6,9,10]

Second, due to non-volatility, NVM maintains the data even after power off. This fact introduces data remanence vulnerability. If an NVM is physically removed from the system, an attacker can directly read out the data from the DIMM. Thus, memory encryption is necessary to enhance security of NVMs.[21-24]

However, memory encryption in NVMs aggravates the write endurance problem, since encryption algorithms usually introduce diffusion property in which a single bit modification in plaintext makes several bit change in ciphertext. [21-23] The diffusion property makes existing bit-level write reduction techniques ineffective for encrypted NVMMs, as they only write the modified bits in a cache line to NVMM by comparing old and new data while encryption causes half of the bits to be flipped. Hence, existing techniques cannot achieve significant data reduction for encrypted NVMM.

Based on these three facts, the paper proposes to revisit the way deduplication and encryption techniques work in current systems.

2.2. Threat Model

The paper tries to secure NVMMs against two physical access based attacks, stolen DIMM and bus snooping. In the former, an attacker steals the NVM and by direct access reads all the data from the DIMM. In the latter, the attacker inserts a bus snooper or a memory scanner in the bus and listens to the data communication between the processor and memory.

2.3. Solution and Design

The paper uses this fact that on average 58% (up to 98% in some cases) of written lines are duplicated, when they examined 20 applications from SPEC CPU2006 [27] and PARSEC 2.1 [28] benchmark suites, as it's shown in Figure 1. This fact calls for performing cache-line-level deduplication on NVMM. The positive outcomes of this includes removing the write latency off from the critical path and speeding up read

and non-duplicate write requests by reducing their waiting time. The first challenge is, in-line deduplication in NVMM should be performed without the decrease of system performance; second, deduplication and NVM encryption should be combined while delivering good performance. The paper proposes DEWRITE as a solution for these challenges.

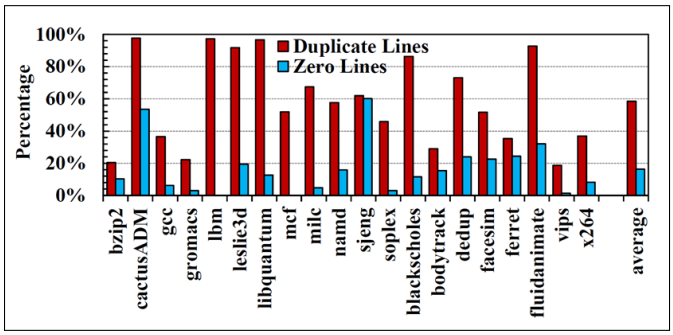

Figure 1: The percentage of duplicate lines. (The first 12 applications are from the SPEC CPU2006 and the following 8 applications are from the PARSEC.)

The paper proposes DEWRITE to do cache-line-level deduplication on NVMM. For deduplication, DeWrite computes the light-weight hash (CRC-32) of a cache line. If the hash of the cache line matches that of an existing line in NVMM, DeWrite reads the line and compares the corresponding data to confirm duplication. DeWrite is able to do the deduplication and encryption serially. This scheme is not efficient for non-duplicated data as the latency of encryption is added to the latency of detection of duplication as is shown in Figure 2 part (a). DeWrite can also performs deduplication and encryption in parallel. This scheme adds the overhead of unnecessary encryption for duplicated data which must be discarded after the result of deduplication, as it is shown in Figure 2 part (b). Thus, a prediction policy will help to select the first mechanism for duplicated data and the second one for non-duplicated data. If a cache line is predicted to be non-duplicate, DeWrite detects duplication in parallel with data encryption to reduce write latency. Otherwise, DeWrite detects duplication without encrypting data to save the computation and energy overhead from encryption.

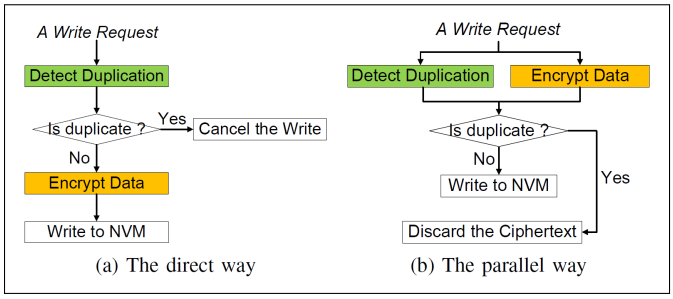

Figure 2: Integrating deduplication and encryption based on prediction for duplication of data.

DeWrite maintains a 3-bit (for the last three recent writes) history window for the whole main memory to predict whether a cache line to be written into NVMM is duplicate ('1' for duplicate,'0' for non-duplicate). If the number of '1's is larger than that of '0's in the three most recent memory writes, the prediction result is duplicated. Otherwise, the result is non-duplicated.

The hardware architecture of DeWrite is shown in Figure 3. In order to reduce the metadata accesses to NVMM on counter mode encryption, they extend the memory controller to include a write-back metadata cache used for buffering the counters. DeWrite uses this metadata cache to prefetch and buffer the deduplication-related metadata and counters. A memory region in the encrypted NVMM is used to store the counters and the metadata of deduplication and encryption. Moreover, DeWrite embeds the per-line counters into the null locations in the data structures to reduce the space overhead from counter storage.

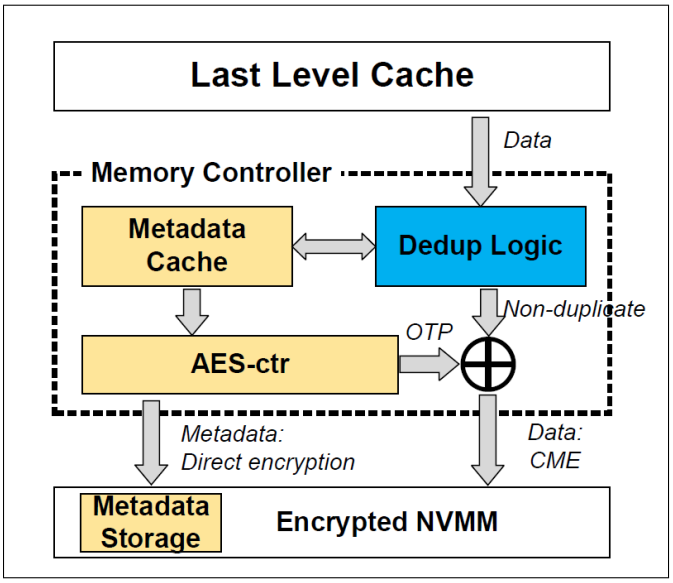

Figure 3: The hardware architecture of DeWrite.

2.4. Evaluation

Compared to the secure NVM system that uses the counter mode encryption without deduplication, DeWrite removes 54% of writes in secure NVMM on average. It speeds up the memory writes and reads of secure NVMM by 4.2× and 3.1×, respectively. It reduces the energy overhead by 40% on average, and it has 6.25% metadata storage overhead.

3. Morphable Counters

3.1. Problem Description

Current security techniques in main memories against physical attacks, introduce significant overhead by additional mem-

ory fetches they need on each data access as follows: (1) a counter needs to be fetched for data encryption, (2) a cryptography hash of data (MAC) is fetched for verifying data integrity, (3) an integrity tree is traversed to prevent replay attacks. All of these steps require multiple additional memory access which downgrades the performance of the system.

Prior works optimize integrity tree traversal by caching tree entries in on-chip caches [7], [11], [12]. However, they neglect the large size of the integrity-tree. Moreover, prior work [14] proposed split counters for encryption, to accommodate more counters per 64-byte cacheline-size tree entry. However, it is not practical to store more than 64 counters per cacheline by reducing minor counter size, since smaller counters can overflow frequently. On an overflow, all the n-minor counters in an entry are reset after incrementing the major counter. This requires extra memory reads and writes for re-encrypting n-child data cachelines on an encryption counter overflow, and for updating hashes of n-child entries on an integrity tree counter overflow. This can cause significant slowdown.

3.2. Threat Model

The paper considers physical access based attacks to the system. It assumes the processor to be trusted, with the off-chip main-memory and memory-bus vulnerable to unauthorized reads, modification or replay attacks. Thus, providing memory security requires (1) data protection via counter-mode encryption, (2) data integrity with Message Authentication Codes (MACs), (3) replay attack protection using integrity-trees.

3.3. Solution and Design

The paper indicates existing counter based encryption techniques usually use 64 same-size counters per cacheline. However, they report that by analysis of overflowing counters on 28 memory intensive workloads from SPEC2006 and GAP benchmark suites, they observed that applications either use less than a quarter of the counters in a cacheline or use all the counters in a cacheline. The distribution of number of counters used in the application is shown in Figure 4.Based on this observation, the paper proposes Morphable Counters (MorphCtr), that changes the counter representation based on the usage pattern to allow; first, providing more than 64 counters in a cacheline; second, avoiding the re-encryption overheads when counters overflow.

They assume at most 128 3-bits counter can be provided. When 64 or less counters are used out of 128 counters in a cacheline, MorphCtr uses a representation called Zero Counter Compression (ZCC) that allocates the entire bit space to only non-zero minor counters in a cacheline. ZCC has a bit-vector to track the non-zero minor counters in a cacheline. In this scheme, MorphCtr supports large counters which overflow less frequently. With 16 non-zero counters,

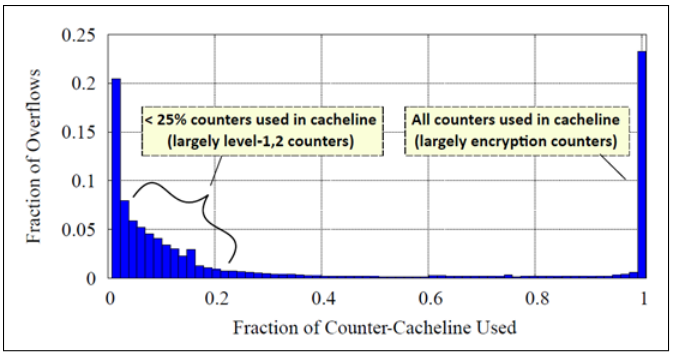

Figure 4: Histogram of fraction of counter-cacheline used when an overflow occurs for SC-64, averaged over 28 workloads from SPEC2006 and GAP.

each counter gets 16-bits, with 32 non-zero counters, each gets 8-bits and when more than 64 out of 128 counters are used, it uniformly allocates 3-bits per minor counter. Figure 5 shows the counter cacheline structure in MorphCtr scheme. MorphCtr represents each counter as the summation of a common base counter and an offset counter. When an offset reaches its maximum, the counters are not reset, instead the base is increased by the smallest offset and all offsets are subtracted by the smallest offset. This method will avoid changing other minor counter effective value, thus prevents subsequent re-encryption overheads. Figure 6 shows this mechanism versus the existing designs.

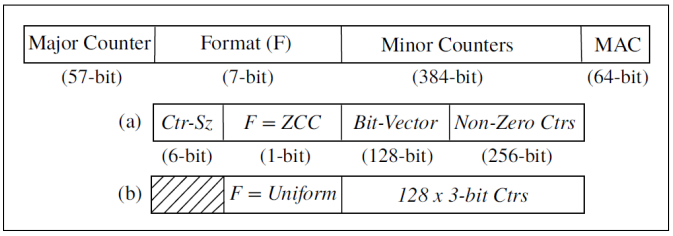

Figure 5: Structure of Morphable Counter cacheline. Counters are represented in (a) Zero Counter Compression or (b) Uniform formats.)

	Major Counter	Minor Counter (3-bit)				Effective Value (*major + minor*)			
Overflowing Minor Counter	100	5	6	8	7	105	106	108	107
After Overflow (Existing Design)	108	0	0	0	0	108	108	108	108
Avoiding Overflow With Rebasing	105	0	1	3	2	105	106	108	107

Figure 6: Avoiding overflow and re-encryption with minor counter re-basing. Existing design resets all minor counters, requiring re-encryptions (all effective values changed); Re-basing only changes effective value of overflowing counter.)

3.4. Evaluation

They evaluate Morphable Counters with 28 memory intensive workloads from SPEC2006 and GAP benchmark suites and compare the design with a baseline using split counters with 64 counters per cacheline (SC-64) for both encryption and integrity tree. Using MorphCtr-128 improves performance by 6.3% on average (up to 28.3%) and reduces energy-delay product by 8.8% on average. These benefits come without any extra storage or reduction in security. For a 16 GB memory, the SC-64 baseline requires an integrity tree that is 4MB in size (4 levels), while the 128-ary tree using MorphCtr is only 1MB (3 levels).

4. Osiris

4.1. Threat Model

The processor chip to be the secure boundary. The attacker can snoop the memory bus, scan the memory content, tamper with the memory content (including rowhammer) and replay old packets. Differential power attacks and electromagnetic inference attacks are beyond the scope of this paper. Furthermore, attacks that try to exploit processor bugs in speculative execution, such as Meltdown and Spectre, are beyond the scope of this paper. Finally, our proposed solution does not preclude secure enclaves and hence can operate in untrusted Operating System (OS) environments.

4.2. Observations

The design of the Osiris are based on these several observations, as follows.

Observation 1: Losing encryption counter values makes reconstructing the Merkle Tree nearly impossible. One possible way to reduce reconstruction time is to employ stop-loss mechanisms that limit the number of possible counter values to verify for each counter after recovery.

Observation 2: Co-locating the data blocks with a few bits that reflect most-recent counter value used for encryption can enable fast-recovery of the counter-value used for encryption

Observation 3: When the ECC function is applied over the plaintext and the resulting ECC bits are encrypted along with the data, the ECC bits can provide a sanity-check for the encryption counter.

4.3. Design

Osiris chiefly depends on surmising the correctness of an encryption counter by calculating the ECC associated with the decrypted text and comparing it with that encrypted and stored with the encrypted cacheline.

Therefore when there is an error detected due to a mismatch between the expected ECC and the stored ECC (obtained after

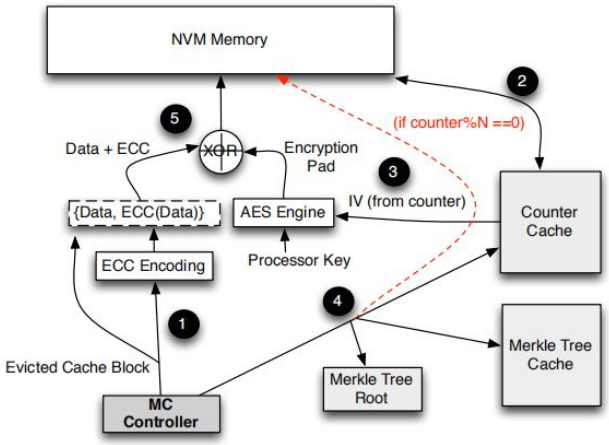

Figure 7: Osiris Write Operation.

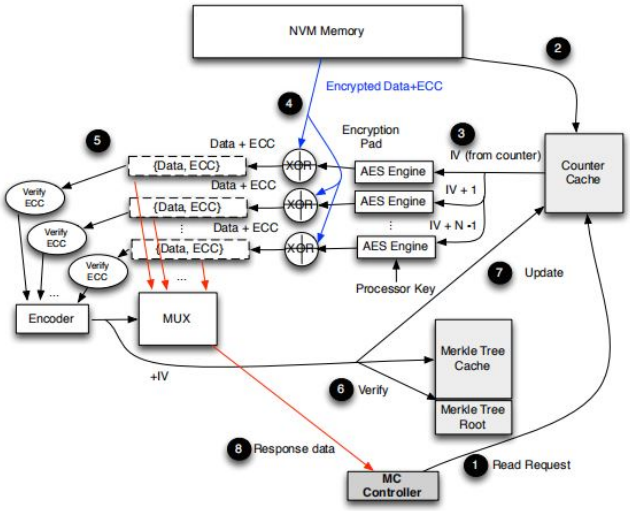

Figure 8: Osiris Read Operation.

decryption) one reason for such an ECC mismatch is using an old IV value. This could be happen because of the counter persistence issue. If a cache block is updated in memory, it is also necessary to update and persist its encryption counter, for both security and correctness reasons. Now we have the ability to detect the use of stale counter/IV, therefore we can implicitly reason about the probability of losing the most-recent counter value due to a sudden power loss or system crash. 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).

Osiris deploys the stop-loss mechanism to limit such possibility to only N updates of a counter, i.e., the correct IV should be within [IV + 1, IV +N], where IV is the most recent IV that was stored/persisted in memory. Note that once the speculated/chosen IV passes the first check through ECC sanity-check it also needs to be verifiified through Merkle Tree.

Osiris Write Operation. As shown in Figure 7, during a write operation, once a cache block is evicted from LLC, the memory controller will calculate the ECC of the data in the block, as shown in Step 1 . Note that write operations happen in the background and are typically buffered in the write pending queue. In Step 2 , the memory controller obtains the corresponding counter in the event of a miss and evict/write back the evicted counter block, if dirty. The counter value obtained from Step 2 will be used to proactively generate the encryption pad, as shown in Step 3. In Step 4, the counter value will be verified (in case of miss) and the counter and affected Merkle Tree (including the root node) are updated in the Merkle Tree cache. Unlike a typical write-back counter cache, if the new counter value is a multiple of N or 0, then the 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.

Osiris Read Operation. As shown in Figure 8, the memory controller will obtain the corresponding counter value from the counter cache (in the case of hit) or memory (in the case of a miss) and evict the victim block, if dirty, as shown in Steps 1 and 2. The counter value will be used to generate the encryption pad as shown in Step 3. In Step 4 , the actual data block is read from memory and decrypted using the pad generated in Step 3. In Step 5, traditional ECC checking occurs. Finally, if the counter block is fetched from memory (miss), the integrity of the counter value is verified, as shown in Step 6. Finally, as shown in Step 7, the memory controller receives the decrypted data, which is then forwarded to the cache hierarchy by the memory controller.

4.4. Reliability and Recovery from Crash

As shown before, to recover from a crash, Osiris and Osiris-Plus must first recover the most-recent counter values utilizing the post-decryption ECC bits to find the correct counters. The Merkle Tree will then be constructed and the resulting root will be verified and compared with the root kept in the processor. While the process seems reasonably simple, it can get complicated in the presence of errors. Specifically, uncorrectable errors in the data or encryption counters make generating a matching Merkle Tree root impossible, which means that the system is unable to verify the integrity of memory. Note that such uncorrectable errors will have the same effect on integrity-verified memories even without deploying Osiris. Uncorrectable errors in encryption counters protected by a Merkle Tree can potentially fail the recovery process. Specifically, when the system attempts to reconstruct the Merkle Tree, it will fail to generate a matching root. Furthermore, it is not feasible to know which part of the Merkle Tree causes the problem. Only the root is maintained; all other parts of the tree should generate the same root or none should be trusted.

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

4.5. Evaluation

As shown from Figure 9, for most of the benchmarks, Osiris-Plus outperforms all other schemes in performance. Meanwhile, for most benchmarks, Osiris performs close to WB scheme. This is because the 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. However it is not common to have the same counter updated many times before eviction from counter cache; once a data cache block gets evicted from the cache hierarchy, it is very unlikely that it will be evicted again very soon. However, since Osiris-Plus can additionally eliminate premature (before Nth write) counter evictions, it can actually outperform WB. The only exception we can observe is Libquantum. That is mainly due to its repeated streaming behavior over a small array (4MB array), leading to many cache hierarchy evictions due to conflicts. However, since each counter cache block covers the counters of 4KB page, many evictions/writes of the same data block will result in hits in the counter cache. Accordingly, a WB scheme performs better as there are few evictions (write-backs) from counter cache compared to the writes due to persistence of counter values at each Nth write in Osiris and Osiris-Plus. However, we observe that with larger limit value, such as 16, Osiris-Plus clearly outperforms WB (7.8% vs. 14% overhead), whereas Osiris performs similarly.

In summary, for all benchmarks, Osiris and Osiris-Plus perform better than strict-counter persistence (WT) and very close or even better than battery-backed WB scheme.

5. Anubis

5.1. Threat Model

Threat model only trusts the processor chip. An attacker can possibly scan the memory, snoop the memory bus, replay memory packets, and can tamper with memory contents or memory bus packets. Attacks such as access pattern leakage, memory timing, power analysis, speculative execution and

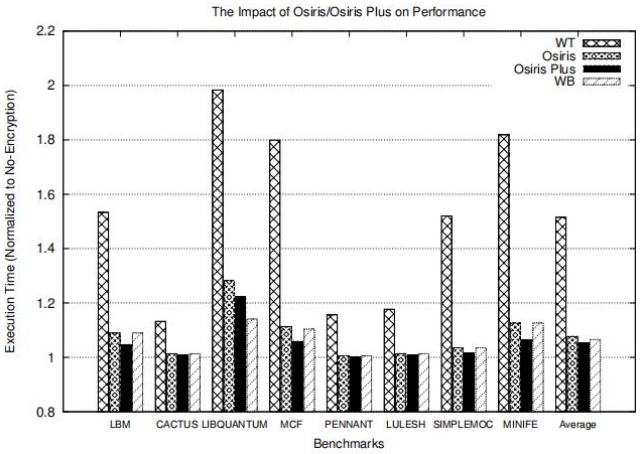

Figure 9: The impact of osiris and osiris plus persistence on performance.

electromagnetic (EM) side channel attacks are beyond the scope of this paper.

5.2. Recovery of Counters

They use Osiris to recover leaves (in case of general Trees). The paper goal is to speed up recovery time by reducing the number of counters that needs to be recovered instead of how they are recovered.

5.3. Merkle Tree Reconstruction

Once all of the counters are corrected using Osiris, or other schemes, the Merkle Tree can be regenerated and the integrity of the counters can be verified by matching the root with the root of the tree stored securely inside the chip. However, rebuilding the Merkle Tree is required to verify the first access; verifying each level would require ensuring that all its children are up-to-date, and hence a recursive operation of calculating MACs over all the tree will be needed after recovery.

5.4. Atomicity of Data and Security Metadata Updates

In modern processor systems, there is a feature called ADR that allows enough power to flush the contents of the Write Pending Queue (WPQ) inside the processor to the NVM memory once a crash occurs. However, the WPQ size is limited tens of entries. Since anything gets inserted in WPQ is guaranteed to persist, i.e., in the persistent domain, we have to ensure that the content of WPQ ensures consistency or at least there is a way to bring the system to a consistent state after power restoration.This leaves us with a challenge on how to atomically insert updates to data, counter, Merkle Tree nodes and Anubis updates in WPQ. To solve this problem, the proposed solution rely on internal persistent registers to hold all the updates (data, counter, Merkle Tree nodes and Anubis updates) before attempting to insert them individually into the WPQ.

5.5. ANUBIS DESIGN

Tracking Updated Security Metadata

One key observation in the paper is that it is sufficient to persistently track the addresses of the blocks in the Merkle Tree and counter caches to significantly reduce recovery time; only the blocks of the tracked addresses have been possibly updated without being persisted. Thus, by having the ability to identify the addresses of the counter blocks that were in the cache at the time of the crash, we only need to iterate through their corresponding counter blocks updated nodes in both, Merkle Tree and counter caches, must be tracked. For terminology, we refer the counter shadow-tracker as Shadow Counter Table (SCT), whereas the Merkle Tree shadow-tracker is called Shadow Merkle-tree Table(SMT).

5.6. Anubis for General Integrity Tree (AGIT)

AGIT Read: Tracking Metadata Reads. As shown in Figure 10(a). only lost nodes and counters need to be fixed. For both caches, the shadow regions are updated on each cache miss, i.e., before reading a metadata block from memory, and hence we call it AGIT Read scheme. Note that such shadow regions are merely used for recovery acceleration; once recovered, as usual, the root will be used to verify all counters and Merkle Tree nodes as they are getting read into the processor chip. Thus, any tampering with the content of shadow regions or unaffected counters (were not in the cache) will lead to root mismatch when the affected (not recovered correctly or tampered with) is read into the processor chip.

AGIT Plus: Tracking Metadata Modifications. As shown in Figure 10(b). In AGIT-Read, SCT and SMT are updated whenever some metadata are brought into the cache disregarding the fact that some metadata would never be modified in the cache. In fact, a significant number of blocks leave the cache without any modification. AGIT Plus reduces extra updates to the shadow tables by acting only whenever metadata is first modified in the Counter Cache or Merkle Tree Cache. This reduces the overhead of AGIT read significantly without hurting the recoverability.

AGIT Recovery Process. The recovery process of AGIT is straightforward. Once the system is booted up upon recovery, the system starts scanning the content of SCT to get the list of possibly lost updates in the cache. For each address, the data blocks (correspond to the possibly lost counters) are read and used to recover counter using Osiris as discussed earlier.

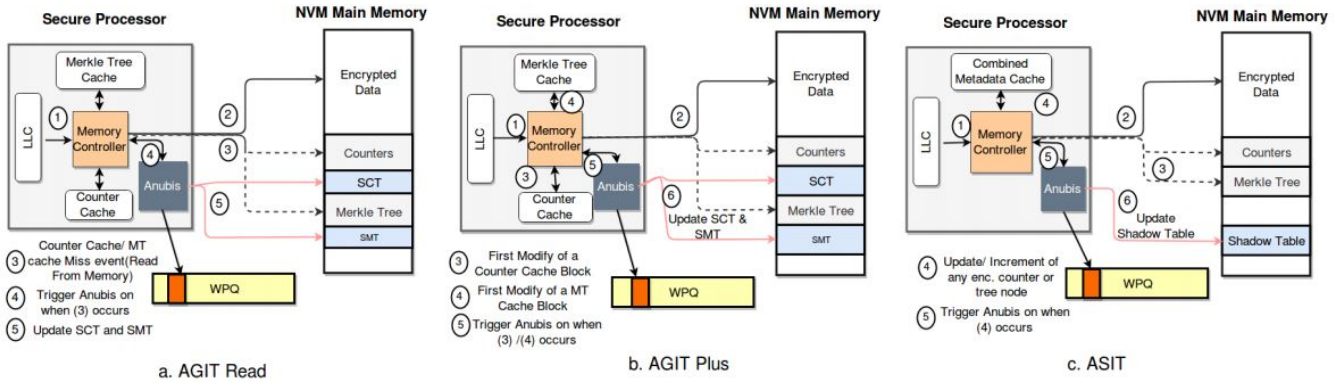

Figure 10: Anubis Operation.

5.7. Anubis for SGX Integrity Tree (ASIT)

Unlike general trees, SGX-style integrity tree advocates for fast updates through limiting dependence between tree levels to only a counter on the upper level. Thus, on each update, affected nodes on different levels can be updated by calculating the MAC over their counters and the updated counter at the upper level.

ASIT Recovery Process. The recovery process in the ASIT scheme is different than that of the AGIT scheme in the following two ways. First, Osiris (or any counter recovery scheme) is no longer needed and hence no need to try different counter values to finish recovery; the LSBs and MAC are replaced directly from the SMT block. Second, instead of rebuilding the Merkle Tree, ASIT only recovers the metadata cache to its pre-crash state.

5.8. Evaluation

Recovery Time: Unlike previous models (e.g., Osiris), the recovery time in Anubis schemes is only the function of the cache size(Counter Cache, Merkle Tree Cache, or Combined Metadata-Cache) and the number of levels in the tree. The recovery time increases with the increase in cache size. In comparison with the state-of-the-art Osiris, the recovery time for 8TB memory is about 28193 seconds(7.8 Hours) while Anubis recovery time for extremely large cache sizes (4MB) is only 0.48s in AGIT, i.e., 58735x faster recovery time.

6. Triad-NVM

6.1. Persistent and Non-persistent Region.

In current systems, at the time of bootup, we can partition the memory into two different region. The kernel can be initialized with a value that determines which range of the memory module is considered persistent and non-persistent. Persistent region can be formatted and mounted as a filesystem, e.g., ext4 filesystem. Additionally, such persistent region can be used by persistency APIs, such as PMDK, which would backup the persistent data structures by files in the persistent region. Any file in the persistent region can be memory-mmaped, i.e., mmaped, and accessed through typical load/store operations. Meanwhile, the non-persistent region in the NVM device is not guarrantee recoverability after a crash; an application will only be able to access data persisted to the persistent region after recovery.

When there is an update for a persistent memory location, all the corresponding parts for such location in Merkle Tree at all levels should be updated up to the root (inside the processor chip) as shown in Figure 11. Updates to Merkle Tree for persistent data should be updating both the memory copy and the cache copy of 1 The root of the Merkle Tree, 2 The root's child node that owns the updated counter, and all other intermediate nodes correspond to the updated counter. Finally, as in 4, updating the counter in both the memory and counter cache. Note that, for non-persistent data, updating the intermediate nodes and counters in the cache and lazily write it back to memory is sufficient. Clearly, the subtrees that belong to persistent regions should be separable, i.e., some parts of the root only belongs to persistent-region and others only to non-persistent regions. Having some parts belong to both is challenging can lead to unverifiable parts after recovery; that part will reflect most-recent values of both types of data, but we will be unable of reproducing the root due to the lost updates of intermediate nodes and counters of non-persistent data.

6.2. Overall Design

Figure 12 depicts the Triad-NVM design. As mentioned earlier, Write-Pending Queue (WPQ) is considered part of the persistence domain (power-fail protected domain) in modern processors. Thus, anything reaches there should be consistent or there is a way to ensure its consistency. To do so, each write operation, before being persisted (removed from volatile buffer), it logs all its corresponding updates (counter, data, MT nodes and root) to persistent registers inside the processor and then set a persistent bit (called READY BIT). If a crash occurs while copying the updates in persistent registers to WPQ, then when the system restores the memory controller will attempt to write the persistent registers to NVM (or WPQ) again. At the time of copying the contents from the registers to

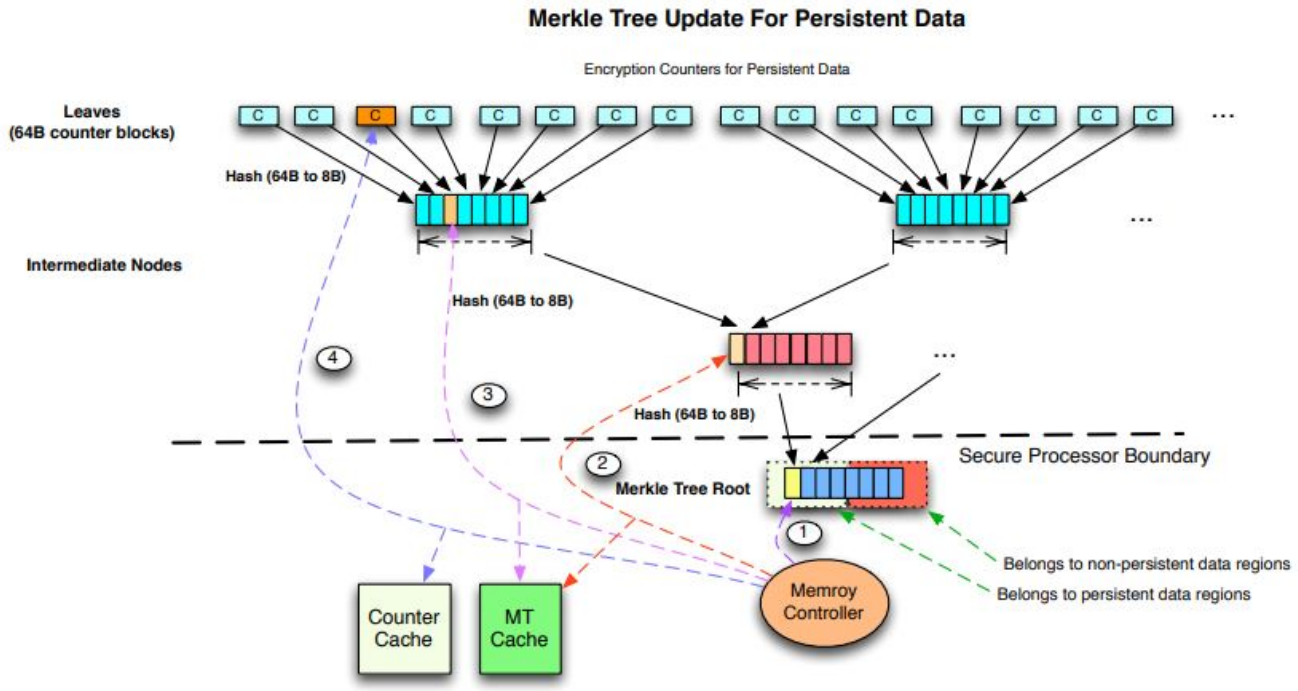

Figure 11: Updating Merkle for counters correspond to data with different persistency requirements.

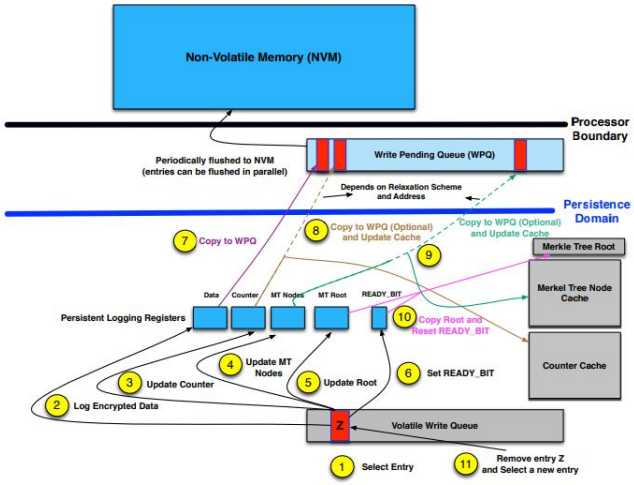

Figure 12: High-Level Overview of Write Operation on Triad-NVM.

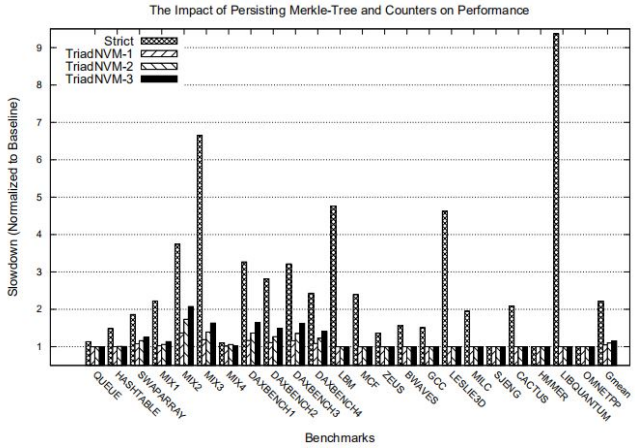

Figure 13: The impact of Merkle Tree persistence model on Throughput.

WPQ, Triad-NVM selectively chooses if the counter or Merkle Tree nodes should be copied to WPQ or it is just enough to update them in caches (as shown in steps 8 and 9). Note that once a dirty block gets evicted from these caches it will go to WPQ as usual. Persistent registers can be implemented as fast NVM registers or volatile registers will be flushed to slower NVM registers once a crash occurs through leveraging ADR or residual power. The number of persistent registers depends on the Triad-NVM model, e.g., if TriadNVM-2 is used then we only need 5 registers, whereas if we use the impractical strict persistence then we might need up to 15 registers.

6.3. Evaluation

The results as shown in Figure 13 show that using strict persistence scheme can lead to an average of 2.21x slowdown, whereas TriadNVM- 1, TriadNVM-2 and TriadNVM-3 lead to only 4.9%, 10.1% and 15.6% performance overheads, respectively. Moreover, the paper observes that write-intensive workloads with non-persistent region allocations, e.g, Libquantum, can get an almost order of magnitude speed up when using schemes that are aware of persistent regions and thus relax the requirements of non-persistent memory ranges.

Table 1: Comparison of Multiple Techniques

Technique	Problem	Threat Model	Solution	Critique
Osiris	Recovery counter	Physical Access (bus snooping and stolen NVM)	Repurpose ECC bits	There is no area overhead estimated
Anubis	Speedup the Merkle Tree reconstruction	Physical Access (bus snooping and stolen NVM)	Create a Shadow Table for tracking the affected tree	Incurs high extra overhead for Intel SGX Style Merkle Tree
TriadNVM	Tradeoff between recovery time and performance	Physical Access (bus snooping and stolen NVM)	Split the security metadata for each region	Requires persisting multiple levels of the integrity tree
DeWrite	NVM limited write endurance and long latency, specially when using encryption	Physical Access (bus snooping and stolen NVM)	Eliminating duplicated writes and parallelizing deduplication and encryption	They have assumed the baseline a very traditional counter based encryption and they have not provided discussion on how their design will fit with commercial products like SGX.
Morph. Ctr.	Counter overflow overhead	Physical access replay attack	Compress Zero Counters, Flexible in size, Make overflow less freq.	The paper does not discuss about how write propagation is done when the node in the cache is updated and how context switch affects it.

7. Comparison and Critique

We made several comparison to the papers that we have discussed above as shown in the Table 1. All of the paper are addressing the memory encryption and integrity verification problem.

In terms of type of problem we can say: DeWrite improves the performance of secure (NV) main memories . MorphCtr improves the performance of secure main memories. Osiris provides counter recoverability without battery backup. Anubis accelerates Merkle Tree reconstruction from hours to few minutes. Triad-NVM proposes solution for security metadata of a system that containing persistent and non-persistent regions in NVM.

The first two techniques, provides solution based on application write-intensity. As a result, their solution is workload-behaviour based and not necessarily general. The last three papers are providing general solution regardless of workloads' behaviour.

Considering the threat model, all of the papers are having the same attack model (physical attack) and the same trust model. The only difference is that the Morphable Counter addressing the traditional DRAM memory where all of the other papers are discussing about Non-Volatile Main Memory.

Both DeWrite and MorphCtr improve the system performance; however, they gain their performance improvement from different stages. DeWrite does not change any step and structure in encryption method. It just introduces a "data" deduplication scheme that is more compatible with secured NVMs and makes it as parallel as possible with encryption. On the other hand, MorphCtr does not change major part of encryption scheme, but it changes "counter" cache line structure and minor counter representation to be more flexible in size. The Osiris provide the counter recoverablity by repurposing the ECC bits to facilitate the sanity check. After we are able to recover the counter the Anubis provides a solution for accelerating the Merkle Tree reconstruction. Triad NVM are proposing different Merkle Tree and other security meta data for each of the persistent and non-persistent region.

We have critiques for the papers. Osiris-plus introduces several additional AES engine inside the Memory Controller, however there is no discussion about the area overhead imposed by adding several AES engine. Anubis is creating Shadow Table to remember the counter that inside the cache at the time the system crashes, however obviously it will incurs extra write overhead for maintaining the Shadow Table especially for Intel SGX style Merkle Tree. Triad NVM to gain a better performance also requiring to persist several levels of the integrity tree.

One question that may arise is, since DeWrite and MorphCtr both proposed their solution based on observations of workload write pattern, are they vulnerable to any potential old/new side channel or not. In DeWrite paper they indicate that the side channel attacks are beyond the scope of the paper and it aims to protect NVMM from physical access based attacks in which the attacker is not the user of the computer. In MorphCtr paper the authors claim MorphCtr does not leak any information in addition to existing side-channels. Even-though the counter overflow may leak information about the

memory access pattern, but such vulnerability exist in the baseline (split-counter 64-ary) as well by monitoring address-bus or plaintext counter-values. In fact, neither DeWrite nor MorphCtr, make the system "more secure", they just increase the system performance. Regarding reducing the information leakage by the overflown counters frequency, we believe MorphCtr "may" work better than baseline, for two main reasons. First, the size of minor counters are able to be as big as 16 bits (based on write patterns of application), thus they overflow less frequently. Plus, when minor counters overflow, the re-encryption may not be needed. Thus, side channels may not be able to guess the memory access pattern. Second, in baseline when minor counter overflows, the major counter is added by the largest minor counter (e.g. added by 256, if minor counter is 8 bits). However, in MorphCtr in the same situation, major counter is added by the smallest minor counter. This policy allows the major counter in MorphCtr to overflow later than major counter in baseline. As a result, the information leakage in such circumstances can be potentially less probable in MorphCtr, "if the write pattern leads to less than 64 counters ". The other case that we should consider is that, in MorphCtr, if an application write pattern demands for more than 64 counters, then it allocate 128 3-bit counters, i.e. they frequently will overflow (on every 8 data write). The only thing which is different than baseline is, in minor counter overflow, the major counter will be added by a number probably less than 8 (in baseline will be added by 8) which does not seem to be significantly better than baseline. In the end, we believe that if write pattern of an application leads to many zero counters (more than 64), the MorphCtr is as vulnerable as baseline.

8. Conclusion

The proposed method from these paper shows promising result to bring secure non-volatile memory into production used publicly since the performance overhead imposed by adding encryption and decryption is ultra-low and the recovery time have been improved significantly without sacrificing the security aspect.

References

[1] Pengfei Zuo, Yu Hua, Ming Zhao, Wen Zhou, and Yuncheng 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, pp.442–454.

[2] G. Saileshwar, P. Nair, P. Ramrakhyani, W. Elsasser, J. Joao, and M. Qureshi, "Morphable counters: Enabling compact integrity trees for low-overhead secure memories," in 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2018, pp. 416–427.

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

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

[5] A. Awad, M. Ye, Y. Solihin, L. Njilla, and K. Abu Zubair, "Triad-nvm: Persistency for integrity-protected and encrypted non-volatile memories," in International Symposium on Computer Architecture (ISCA), 2019.