Phoenix: Towards Ultra-Low Overhead, Recoverable, and Persistently Secure NVM

Mazen Alwadi, *Student Member, IEEE* Kazi Abu Zubair, *Student Member, IEEE* David Mohaisen, *Senior Member, IEEE* Amro Awad, *Member, IEEE*

Abstract—Emerging Non-Volatile Memories (NVMs) bring a unique challenge to the security community, namely persistent security. As NVM-based memories are expected to restore their data after recovery, the security metadata must be recovered as well. However, persisting all affected security metadata on each memory write would significantly degrade performance and exacerbate the write endurance problem. On the other hand, relying on an encryption counters recovery scheme would take hours to rebuild the integrity tree, and will not be sufficient to rebuild the Tree-of-Counters (ToC). Due to intermediate nodes dependencies it is not possible to recover this type of trees using the encryption counters. To ensure recoverability, all updates to the security metadata must be persisted, which can be tens of additional writes on each write. In this paper, we propose Phoenix, a practical novel scheme which relies on elegantly reproducing the cache content before a crash, however with minimal overheads. Our evaluation results show that Phoenix reduces persisting security metadata overhead writes to 3.8% less than a write-back encrypted system without recovery, thus improving the NVM lifetime by 8x. Overall Phoenix performance is better than the baseline.

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

1 Introduction

Non-Volatile Memories (NVMs) are emerging as promising contenders to the DRAM, and promise to provide terabytes of persistent data capacity accessible using regular load and store operations [16], [17]. Secure NVM systems commonly aim to protect confidentiality, integrity, and availability of the memory. While data persistency is an attractive feature that enables persistent applications, e.g., filesystems and checkpointing, it also facilitates data remanence attacks [3], [8], [26], [27]. To protect the NVM data at rest, encryption becomes a necessity. Encryption targets the confidentiality among the security requirements. However, encrypting the data introduces the overhead of encryption metadata, which needs to be persisted to ensure secure and functional recovery [2], [6], [18], [26]. In state-of-the-art secure processor systems [3], [5], [10], [22], [27], the counter-mode encryption is used due to its security and performance advantages.

Counter-mode encryption is used in state of the art secure processor architectures [5], [6], [22], [23], [26], where each cacheline is associated with an encryption counter that is used along with a processor key to generate a One-Time-Pad to encrypt the cacheline once written to the memory. The confidentiality of such counters is considered unnecessary, however, their integrity must be protected; as encryption counter reuse can facilitate known-plaintext attacks. Merkle Tree is generally used to verify the encryption counters integrity. Merkle Tree is a tree of hashes over hashes where the leaves are the encryption counters, and finally the last resulting single hash is called the root of the tree, which is always kept in the processor chip. Each counter update changes the root value, and hence any tampering will be detected due to root mismatch. Integrity trees have been also deployed in commercial products for secure processors, e.g., Intel's SGX.

In this paper, we aim to provide persistent security, recoverability, NVM friendly, and ensure ultra-low recovery time with minimal performance overhead. Recoverability is perhaps the most promising system feature that NVMs provide. Thus, recoverability should be considered when adding security features (encryption and integrity verification). Moreover, NVM's near-zero idle power consumption makes it very promising for data centers, cloud systems and HPC systems. In such systems, availability is a strict requirement. Our results show that recovering a practical NVM size (8TB) with current secure processor implementations would take 7.8 hours. On the other hand, high-availability systems have stringent requirements of 99.999% (five nines rule), i.e., the system can sustain a total of 7.8 hours down time only once each 89 years. For instance, for each minute of the system being down in Amazon's cloud system, it is estimated to cost 70 thousand dollars per minute [13]. Simply, one can imagine an in-memory database system, where transactions are taking place, and a crash happens right after committing a transaction. In such a case, the whole Merkle Tree must be recovered to ensure the memory integrity, which will prevent any new transactions from taking place until the recovery process is done.

Crash consistency problem is mainly caused by the inconsistency of security metadata with the memory data. In current secure processor architecture implementations, most of the security metadata updates occur in volatile caches inside the processor chip, without strictly persisting them in memory. Thus, once a crash occurs and caches lose their content, the data might be written to memory while its updated encryption counter and MT updates has not been reflected in memory yet. Therefore, during recovery, the system will have an inconsistency between the memory data and its corresponding security metadata. While strictly persisting the

- *M. Alwadi and D. Mohaisen are with the Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL 32826. K. Zubair and A. Awad are with the Department of Electrical and Computer Engineering at North Carolina State University, Raleigh, NC 27695. D. Mohaisen (mohaisen@ucf.edu) and A. Awad (ajawad@ncsu.edu) are the corresponding authors.*

- *A preliminary version of this work appeared at 46th ACM International Symposium on Computer Architecture (ISCA 2019).*

security metadata updates would eliminate the problem, however, at the cost of tens of writes for each data write. Given the limited write-endurance and very slow NVM writes, strictly persisting the security metadata is considered impractical. Meanwhile, since counter and MT caches can hold thousands of security metadata cache blocks [3], [18], solutions relying on providing enough power to flush the caches content are not practical as well. In fact, state-of-the art processors have a limited support to flush few entries that are guaranteed to be persisted by the system in case of a crash. For instance, in recent Intel's processors, a small buffer co-located with the memory controller is called Write Pending Queue (WPQ) [6]. The WPQ is considered a part of the persistent domain, whenever a write reaches the WPQ it is guaranteed to be written to the NVM. The power to flush the WPQ entries to the NVM is provided by the Asynchronous DRAM Self-Refresh (ADR) [1]. Thus, given the limited number of WPQ entries, high costs of uninterruptible power supplies, the demand for battery-free solutions, area, and environmental constraints, it is important to look for new solutions to enable fast recoverable and crash consistent systems [2], [4], [18], [26].

The state-of-the-art scheme, Osiris [26], addresses the crash consistency problem by leveraging the ECC as a sanity check to recover the encryption counter most recent value. However, Osiris would take hours to iterate over the encryption counters and fix them, then it would take more hours to rebuild the whole integrity tree. Moreover, ToC can not be rebuilt using the encryption counters due to intermediate nodes dependencies.

In this paper, we aim to bridge the gap between recoverability and high-performance for secure NVM systems. As such, we propose Phoenix, a novel memory controller design that achieves both recoverability and high-performance of secure NVM systems. Phoenix is based on our observation that to successfully recover the system we only need to recover the dirty cached blocks. Phoenix reconstructs the exact content of the security metadata cache after a crash by tracking the security metadata cache updates. In fact, we can reconstruct the exact lost cache state after recovery by recalculating the potentially lost values and then verify the integrity of the reconstructed cache. By relying on value recovery of the tree leaves only a small subset of updates to the cache needs to be persisted in memory. Meanwhile, we still can verify that the recovered cache content reflects exactly the same cache state before the crash. Our optimization, realized in Phoenix$^+$, relaxes persisting encryption counters on eviction, to only persist encryption counters on the N-th write, reducing Phoenix's overhead significantly.

To evaluate Phoenix, we use Gem5 [7], a full-system cycle-level simulator, and run representative workloads from the SPEC CPU2006 benchmark suite to observe a reduction of write overhead: Phoenix$^+$ is shown to have less writes than write-back scheme, i.e., improves lifetime by ∼8x. Moreover, Phoenix$^+$ has an average execution time that is even less than write-back scheme.

In summary, the contributions of our work are the following:

- We propose Anubis, a novel memory controller design that enables low-overhead and low recovery time for integrity-protected systems.
- We propose Phoenix, a novel memory controller design that recovers the security metadata cache content with minimal performance overheads and minimal writes.
- We propose Phoenix$^+$, an optimization of Phoenix that improves the system performance and eliminates the extra writes required to recover the Tree of Counters.

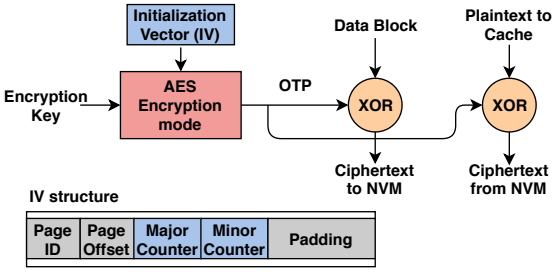

Fig. 1. Counter-Mode Encryption in state-of-the-art secure memories [3], [26].

- Enable crash consistency for secure memory systems and allow recoverability with less than a second.

The rest of the paper is organized as follows. In Section 2, we discuss the background and motivation. In Section 3, we discuss the design of Phoenix. In Section 5, we discuss our evaluation methodology followed by our evaluation. We review the related work in Section 6. Finally, we conclude our work in Section 7.

2 Background and Motivation

In this section, we review background and related concepts, and motivate for our work. In particular, we start by defining the threat model, followed by all relevant concepts.

2.1 Threat Model

Similar to the state of the art [3], [11], [18], [20], [24], [26], our threat model considers the processor chip to be the secure boundary, and to contain the root of the integrity tree and the encryption key, where everything outside the processor is untrusted. We assume an attacker capable of performing passive and active attacks, including bus snooping and replaying memory packets, can scan the memory contents, and may tamper with memory contents. We also assume the attacker can perform attacks while the system is either on or off. Access pattern leakage attacks, electromagnetic (EM) inference attacks, and differential power analysis attacks are beyond the scope of this work.

2.2 Counter Mode Encryption

One of the major security vulnerabilities of NVM systems is the data remanence problem. Therefore, NVM is usually paired with encryption to protect data confidentiality. The state-of-the-art secure processors (e.g., Intel Xeon Processor E-family) use counter-mode encryption, shown in Figure 1, since it provides strong defenses against a range of attacks (e.g., snooping, known plain-text, and dictionary-based). Moreover, the counter-mode has a smaller encryption/decryption overhead compared to other schemes due to overlapped latency of data fetching and one-time-pad generation [3], [5], [27]. For each write to a data block, its' associated counter will be incremented by one. The updated counter is used to generate an initialization vector and then coupled with a processor key serve as inputs for the encryption engine to generate a One-Time-Pad (OTP). After being XOR'ed with this OTP, the data block is considered to be encrypted and can be saved in the memory. Similarly, a read request uses the same encryption pad to generate plain-text for the processor but without updating any counter value.

The size and organization of counters vary in different state-of-the-art schemes. The counters used in Bonsai-Merkle-Tree (BMT)

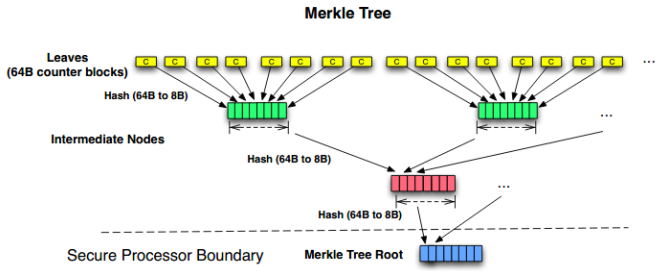

Fig. 2. Merkle Tree

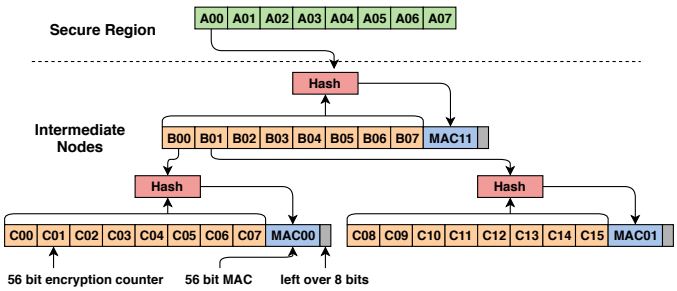

Fig. 3. SGX style Parallelizable Merkle Tree [11]

are organized as per cacheline 7-bit minor counter, and 64-bit per page major counter, each encryption counter (64 Bytes) block can accommodate 64 minor counters and one major counter. Thus, each page will have one encryption counter block. On the other hand, counters used in ToC are ***monolithic counters***, where each of 56-bit long associated with a data block and one 64B counter cacheline can accommodate counters for eight 64B memory blocks. Encryption counter overflow can be costly, and causes long system stalls which is generally unacceptable [20]. Therefore, the monolithic counter should be large enough to prevent overflowing, which means more storage overhead. For encryption/decryption, the monolithic counter will be padded with a block address to generate the initialization vector [20]. To encrypt/decrypt the data, secure processors use AES counter-mode encryption. Figure 1 shows how counter code encryption works.

Several other state-of-the-art schemes use the ***split counter*** scheme [2], [3], [6], [18], [21], [23], [26], in which each data block is associated with one per-page major counter and one per-block minor counter. The major counter is shared by all the blocks within that page. Encryption/Decryption requires knowing both major and minor counter values to generate the OTP. Since each minor counter only accounts for seven bits, and the major counter for 64 bits, a small storage overhead occurs. However, when a minor counter overflows, the major counter is incremented by 1 and the whole page has to be encrypted using the new major counter [2], [3], [6], [18], [21], [23], [26].

2.3 Integrity Verification

Since the trusted boundaries are limited to the processor chip, whenever a block is fetched from the memory, the blocks' integrity needs to be verified. In state-of-the-art research and secure processor designs [2], [6], [20], [26], the Merkle Tree—one of the approaches used for ensuring integrity, is widely studied and used for memory integrity verification. Depending on the tree structure, Merkle trees can be non-parallelizable (e.g., Bonsai Merkle Tree) or Parallelizable (e.g., SGX style counter tree) [2].

Basically, Bonsai-Merkle Tree (BMT) is an N-ary hash tree where the leaves correspond to encryption counters for data blocks [20] and every N leaves will have a hash value calculated based on the values of the counters. Similarly, all the intermediate nodes up to the root are constructed using the hash value based on its children. The root is always kept secure; that is, it never leaves the chip. Moreover, any tampering with a counter leads to the failure of reconstructing the root. Figure 2 shows an 8-ary Merkle Tree, in which the yellow blocks represent the encryption counters, the green and red nodes represent intermediate levels of the MT, the green nodes are the hashes of the encryption counters, and the red nodes are the hashes of the red ones. The process of hashing continues until the root is created, which is shown as the blue node.

Since hashes in the BMT are calculated over the bottom level hashes, the tree updates must be done sequentially. ToC integrity trees, on the other hand, can perform a parallel update of the tree, as the MAC values are not calculated over the below level MAC value. Figure 3 illustrates the organization of ToC integrity tree where each node is comprised of eight counters. The MAC values are calculated over these eight counters and one counter from the parent node as in Figure 3.

2.3.1 Read and Verify

To better understand the verification steps in ToC integrity trees, Figure 3 demonstrates a scenario of verifying counter `C00`. Note that `C00` falls within a block (64 bytes) that contains counters `C00 – C07` in addition to a MAC value. However, verifying `C00` also requires reading `B00` in the upper level, and then calculating the MAC value over `C00 – C07` and `B00`, then compare it with `MAC00`. However, it is important to note that this is assuming `B00` is already verified and cached in the processor chip. However, if `B00` is not present in the processor chip, it must be also verified the same way before we can use it to verify `C00`. The same process of verification is followed in BMT whenever a cacheline is read from the memory.

Clearly, there is an inter-level dependency in the integrity tree, and missing an updated MAC due to a crash can cause the whole recovery process to fail.

2.3.2 Write and Update

To better understand how updates propagate through the integrity tree, let's take the case of updating `C00` in Figure 3. For now, let's assume that there is no expectation of integrity tree recovery after a crash. In its simplest form, updating (incrementing) `C00` requires recalculating `MAC00` after incrementing `B00`. Similarly, `MAC11` will be recalculated with the incremented `B00` and `A00` values. One important aspect to note here is that on each update, the MAC values on the affected nodes can be calculated in parallel using the incremented counter values. In contrast, and for BMT, calculating the upper levels requires the MAC value as an input, hence mandating the serialization of updates (bottom-up). Thus, ToC trees provide parallelism in updating the tree mainly because calculating the values of counters affected on each node can occur in parallel, hence calculating the corresponding MAC values on each affected node.

2.4 Recoverability in Persistent Memories

The security metadata cache caches the integrity tree and encryption counters, and can be eagerly or lazily updated. In the eager

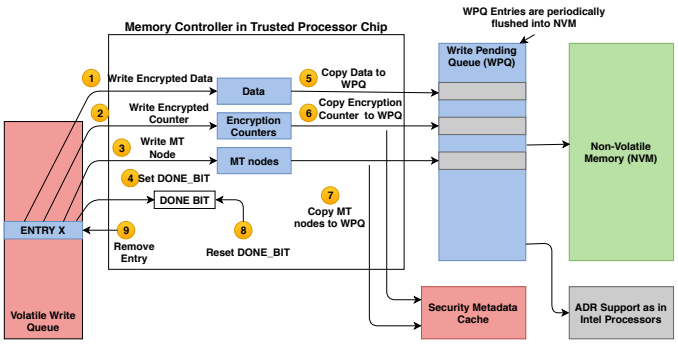

Fig. 4. Atomic Persistence of Integrity-Protected NVMs [2], [6].

update schemes, each write needs to update all the related nodes up until the root. Thus, the root always reflects the most recent state of the tree and can be used to verify the memory integrity after recovery. In contrast, the lazy cache update scheme updates the leaf on each write and relies on propagating the updates upwardly only after the eviction of updated nodes. In the lazy update scheme, the root might still be stale while the metadata cache has the most recent values. Therefore, the lazy update scheme is expected to be used in systems with no recovery expectation [2], [6].

In Merkle tree schemes where the tree can be regenerated using only the leaves (e.g., BMT), it requires a long time to rebuild the tree. Once the tree is reconstructed, the generated root will be compared against the one inside the processor chip which has been eagerly updated. In contrast, the lazy update scheme has no way to verify the integrity of the reconstructed tree since the root is out-of-date; the root does not reflect the most recent changes to memory before the crash. Moreover, in the ToC integrity tree, it is impossible to regenerate the previous state of the tree from the leaf encryption counters since every intermediate node contains eight versions, the updated value of which could be lost during a crash. Due to such inter-dependency of levels, and the use of volatile metadata cache, it is very difficult to recover systems with ToC even if an eager update scheme is maintained.

While an eager update is suitable to rebuild the Merkle tree using only the encryption counters, it is not the case with ToC integrity tree. In ToC integrity tree, each node contains a MAC value calculated over the node counters and a nonce from the parent node. This inter-dependencies makes it very complex to retrieve the lost intermediate nodes during the recovery process. In a lazily updated ToC integrity tree system, the root is not enough to verify the integrity of the memory as it might be stale. Thus, to verify the integrity of the memory the integrity tree should be restored, while each node is used to verify the integrity of lower and upper levels.

2.5 Counter Recovery Schemes

Prior work on general Merkle Tree [6], [26] explored how to recover encryption counters after a crash. One of the proposed solutions, Osiris [26], relies on encrypting Error-Correcting Code (ECC) written with data. By limiting the number of updates to a counter before persisting it to the memory, e.g., every 4th write, it can recover the counter used to encrypt the data by relying on the fact that a large number of errors will be detected by ECC when a wrong counter is used. By trying multiple counter values, Osiris can recover the counter used to encrypt the data. For more details on Osiris, the reader is referred to [26]. While Osiris presents a novel approach that reduces the overhead of persisting counters significantly, there are many other competing approaches [18]. For instance, as also discussed in [26], part of the encryption counter used for encryption can be also written with the data and thus strict the persistence of the whole encryption counter can be relaxed. For the rest of this paper, we assume Osiris can be used, however, any other counter recovery scheme would work.

2.6 Atomic Updates of Security Metadata

While persisting the security metadata allows the system to recover after a crash, if a crash happens before persisting the security metadata of a a persisted data block, the memory will have an inconsistent security metadata which will not be able to decrypt/verify the data. To ensure the security metadata is consistent with the data, the update should be done atomically. Modern processors provide enough power to flush the content of the Write Pending Queue (WPQ) when a crash occurs, and the power to flush the WPQ content is provided by the Asynchronous DRAM Refresh (ADR) feature [6]. Therefore, all writes that have reached the WPQ are considered to be persisted. Additional bits, such as READY_BIT or DONE_BIT can be used to ensure the content of persistent registers are inserted atomically to the WPQ [6], [18]. Figure 4 shows how atomic updates are done, the encrypted data, encryption counter, and the updated MT nodes are moved to persistent registers, then the DONE_BIT is set. After that the updates are moved atomically to the WPQ, then the data is written to the NVM, and finally the DONE_BIT is reset and the entry is removed from the WPQ.

2.7 Motivation

As discussed earlier, parallelizable Merkle Trees, as in SGX-style trees, are very challenging to recover. Unlike typical Merkle Tree, the verification of each level counters also relies on the cryptographic hash values stored in their children nodes; the hash values in children levels are calculated over the counters in that child node and a counter in the upper level, i.e., each upper-level counter is used as a version number that will be verified by the children hash value and later verified by the hash value of its neighboring counters along with its parent counter. The process continues until all corresponding parent counters are verified, however, if such parent counters exist in the cache, it means that they are already verified and thus it is sufficient to stop once the lowest level hit in the cache is found. Unfortunately, if any of such intermediate nodes get lost during a crash, even if the root is saved inside the processor, it is impossible to verify the integrity of the leaves; they strictly rely on verifying all their parent counters. Thus, due to this inter-level dependence in this style of Merkle Tree, it needs special handling to enable their recovery. Meanwhile, general Merkle Tree implementations, such as Bonsai Merkle Tree, can be completely reconstructed if we can recover encryption counters (the leaves) as explained in prior work [26]; upper levels of Merkle Tree are simply the hash values of lower-levels, and thus as long as the root hash value matches after reconstruction, the counters are considered verified. However, even for general Merkle Tree, the recovery time is impractical for practical NVM capacities, e.g., 4TB or 8TB [6]. Figure 5 shows the recovery time for different NVM capacities assuming encryption counters can be recovered using state-of-the-art counter recovery scheme [26]. Counter recovery schemes rely on reading data blocks to use their accompanied ECC bits as a

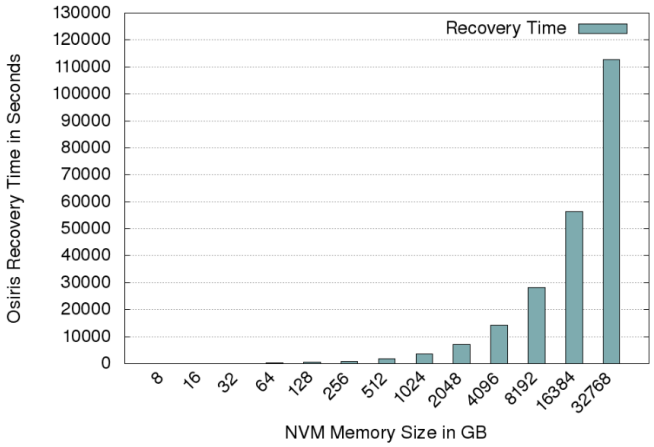

Fig. 5. Recovery time using Osiris.

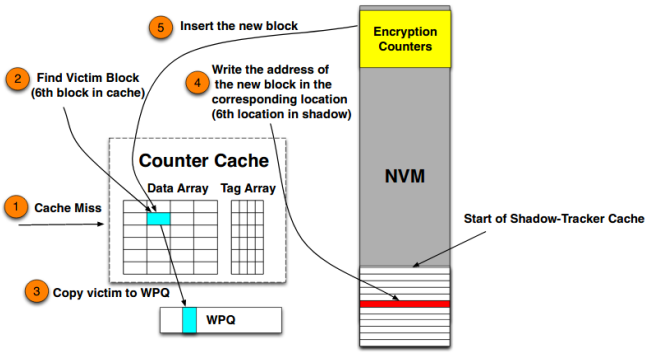

Fig. 6. Shadow Table

sanity-check to recover the used encryption counter [26]. Thus, the recovery time scales linearly with the size of the memory (number of data blocks). For instance, we can see that at 8TB NVM capacity, almost 8 hours are needed to recover the system. With the expected huge capacities of NVMs, any operation that is O(n), where n is the number of data blocks (typically 64B) in memory, can take impractical time and should be avoided.

For instance, for an 8TB memory system, strict persistence needs to persist additional 13 writes on each regular memory write, i.e., reducing NVM lifetime and increasing write bandwidth by 13x. Clearly, strict persistence is impractical. Anubis brings down the overhead of strict persistence significantly, although it is still high. In particular, Anubis incurs 2x the write bandwidth by persisting each update to cache in the shadow region. Thus, Anubis reduces the lifetime of NVM systems to almost half of its actual lifetime, although the lifetime of NVMs is already short, to begin with. Moreover, NVM writes are slow and power hungry, hence can significantly degrade the performance and increase the overall power consumption. Figure 5 shows the overhead of Anubis scheme, which can limit its deployment, and motivates for this work. In particular, Figure 5 shows the impact of Anubis on the number of writes. On average Anubis, incurs almost 2x the number of writes and average performance overhead of 7.9% compared to baseline secure NVM without recovery support. The goal of Phoenix is to provide an NVM-friendly solution that does not incur significant NVM writes. Thus, Phoenix is proposed as a practical solution that realizes low-overhead secure and recoverable NVMs.

3 Anubis Design

3.1 Tracking Updated Security Metadata

One key observation we have 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. Similarly, by tracking the addresses of the Merkle Tree nodes in the Merkle Tree cache, after fixing lost counters (using Osiris [26]), reconstructing general Merkle Tree can be implemented by starting from leaves, fixing those identified as lost, then going to the upper level and fix those identified as lost, continuously until reaching the top level. The order of fixing is important; repairing upper levels relies on fixing lower levels first. In its simplest form, shadow-tracking can be implemented by reserving the cache size in NVM, e.g., a 128KB will be reserved in NVM for shadowing the addresses in the 128KB counter cache. During counter cache miss event, based on the location of the victim block in the data array of the counter cache, the address of the new block will be written to NVM on the offset corresponds to the location in the cache, as explained in Figure 6. Similar approach can be used for shadow-tracking Merkle Tree. Note that the position of the block in the counter cache remains fixed for its lifetime in the cache; LRU bits are typically stored and changed in the tag array. Since the miss rate of counter cache is typically very small, the additional writes will be minimal. As mentioned earlier, 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), where as the Merkle Tree shadow-tracker is called Shadow Merkle-tree Table(SMT). In both cases, the storage overhead is minimal, e.g., for a 128KB counter cache size and 8TB memory, the overhead is only 128KB/8T.

3.2 Anubis for General Integrity Tree (AGIT)

3.2.1 *AGIT Read: Tracking Metadata Reads.*

As discussed earlier in Section 2.4, when eager cache update scheme is used, general Merkle Tree can be recovered by restoring the leaves and updating the tree upwardly before finally verifying that the resulting root matches that inside the processor. Thus, we can directly apply the idea of shadow-tracking for both Merkle Tree and counter cache to speed up the recovery process; 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 in to the processor chip.

Figure 7(a) illustrates the operation of AGIT Read. As shown in the figure, once a memory request arrives at the memory controller (Step ①), the required encryption counter and Merkle

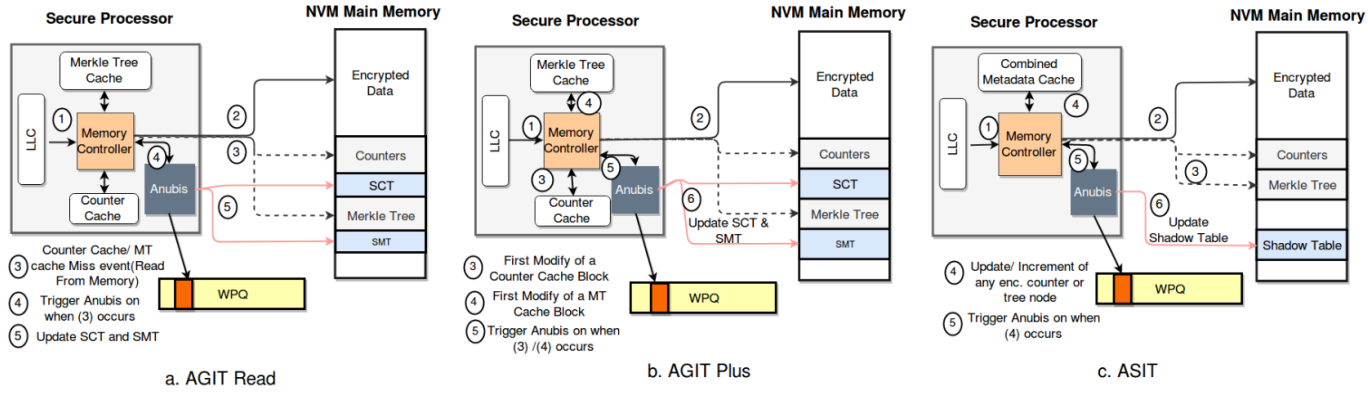

Fig. 7. Anubis Operations

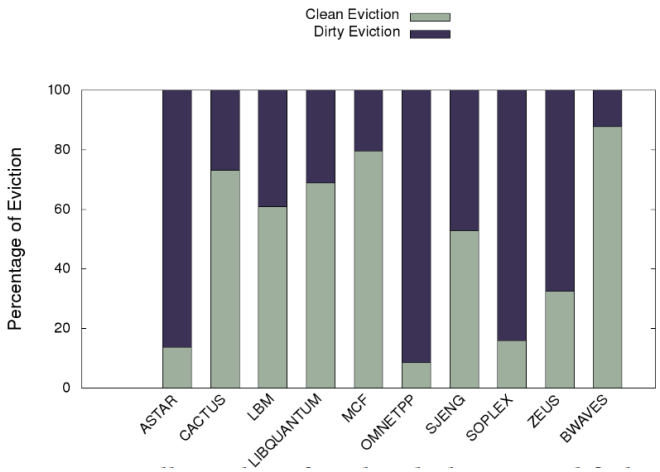

Fig. 8. Cache block modifications

Tree nodes are retrieved from memory (as shown in Steps ② and ③), in case not present in the cache. Before inserting any of the counters or Merkle Tree blocks in the shadow cache, Anubis prepares the address blocks (used for tracking) of the counter and Merkle Tree blocks and insert them into the WPQ (as shown in Step④) before inserting them in the cache (in Step⑤). Note that since the tracking blocks (prepared in Step④) are already in WPQ (persistent write queue), they are considered persistent, however, they will be eventually written to the SCT/SMT region in memory as soon as WPQ is flushed or such entries get evicted from WPQ.

3.2.2 AGIT Plus: Tracking Metadata Modifications.

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. As illustrated in Figure 8, most applications evict a large number of cache-blocks from the counter cache that are clean. Hence, only tracking addresses of modified blocks provide the same recoverability but with reduced overhead. Moreover, most dirty cache-blocks are updated multiple times in the counter cache and Merkle Tree cache. In fact, Merkle Tree cache blocks reside in the cache and get modified more than counter cache blocks. Only tracking once during a dirty cache block's lifetime is sufficient to successfully recover the system. Based on these observations, 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 recover-ability. AGIT-Plus (as shown in Figure 7(b)) is similar to AGIT-Read except that it triggers Anubis only at the first update to a counter or Merkle Tree blocks in the cache (as in Steps③ and ④), i.e., setting the dirty bit for the first time. Before completing the update to caches, the generated shadow blocks must be inserted in WPQ.

3.2.3 AGIT Recovery Process.

Algorithm 1 AGIT Recover Algorithm

```
Read SCT and SMT
for SCTi in SMT do
    Read Counter Block at address stored in SCTi
    for SCTi in SMT do
        Counter in Counter Blocki
        Fix Counter using Osiris
    end for
end for
Classify SMT entries based on their level in tree
MaxLevel ←Total Merkle Tree Levels
Affected Nodes m ←Total Affected Nodes at Level m
m ←0
for m<= MaxLevel do
    for Node in Affected Nodes m do
        Read all child nodes of Node
        Create hash of child nodes and replace Node
    end for
    m ←m+1
end for
IF Stored root matches new root Then
System restored
ELSE
System unrecoverable
```

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 the counter using Osiris as discussed earlier. Note that any other counter recovery scheme would likely have to read the data block, e.g., using phases or

extending data bus. Later, once the affected counters are fixed, AGIT scans through SMT to search for possibly lost updates in the first level (immediate parents of counters) and recalculate the nodes' values based on their immediate children (counters). Later, once the 1st level is fixed, AGIT scans SMT for possibly lost updates to the second level and fix them by calculating their values based on their immediate children (level 1). AGIT proceeds with the same process by going up in the tree level by level and eventually reach the root of the tree. Once at the root level, the resulting root from the calculated tree will be compared against what is in the processor chip to find out if the recovery has been successful. If the resulting root mismatches what is in the processor chip, then the recovery process has failed, and the system raises a warning about that. Note that the speed up in recovery mainly stems from the fact that we only need to fix the lost counters and Merkle Tree nodes than naively iterate through all the blocks as in systems without Anubis.

3.3 Anubis for SGX Integrity Tree (ASIT)

Unlike general trees, SGX-style integrity tree advocates for fast updates by limiting dependence between tree levels to only a counter on the upper level. Thus, on each update, affected nodes on different levels are updated by calculating the MAC over their counters and the updated counter at the upper level [11]. However, this comes with extra complexity during reconstruction after a crash. Each intermediate node depends on a counter on the upper level, and counters of each level are verified using the MAC value co-located with each node. Thus, by losing the MAC values on different levels, it becomes infeasible to verify the integrity of the tree. Meanwhile, reproducing the MAC values of the intermediate tree is not safe until the counters of the level are verified. In SGX, 8 of the 56-bit encryption counters are stored along with a 56-bit hash in one single cache line. Each parent node also contains 8 counters (56-bit each), and a 56-bit hash value. However, as mentioned earlier, it is very challenging to recover the tree to its previous state after a crash and most of the time quite impossible if some intermediate nodes in the tree are missing. ASIT aims to provide a book-keeping mechanism that tracks the tree during runtime and recovers after a crash very quickly. Since encryption blocks in SGX have a similar structure to intermediate levels, a single metadata cache is typically used. For the shadow table, we also merge the SCT and SMT into one larger Shadow Table (ST) with a size similar to metadata cache. Figure 9(b)shows the organization of the Shadow Table for ASIT scheme.

3.3.1 ASIT Metadata Tracking.

ASIT's main idea is inspired by Anubis's embalming capability. In particular, ASIT aims to have an exact persistent copy of the content of metadata cache before the crash. However, such a shadow copy must have its integrity protected against any possible tampering. By doing so, it is sufficient to just restore the metadata cache by copying back the shadow cache after verifying its integrity. In ASIT scheme (Figure 7(c)), on each update to encryption counters (due to write requests) in the cache, the Shadow Table (ST) is updated with the modified cachelines in the cache. As described earlier (Section 2.4), an eager update scheme is inappropriate for SGX-style trees; having a root value that reflects the most recent tree is insufficient to recover the tree. Meanwhile, strictly tracking all changes to counters and Merkle Tree would incur significant overheads with eager update scheme;

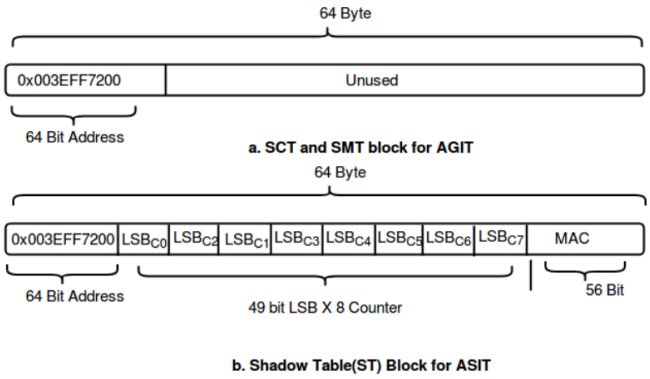

Fig. 9. Shadow Table Blocks

each write would incur 12 writes to the shadow region. Therefore, we opt for using lazy update scheme while strictly tracking the changes to the metadata cache; in the lazy scheme only one block is typically updated on each memory write, and thus it is more practical to track such updates compared to eager update scheme. However, the cost is the need to fully protect the integrity of the shadow-table through a small general Merkle Tree (3-4 levels) with its root be persistent and never leaves the processor chip. The updates to the tree protecting the shadow table use eager cache update scheme, i.e., the root of the shadow table tree reflects the most recent state of the shadow table. Each Shadow Table block contains the following elements (Figure 9(b)):

- **Address:** 64-bit address of the Merkle tree block/encryption counter block modified in the combined metadata cache.
- **MAC:** 56-bit MAC value calculated over the updated counter values (nonces in that node).
- **Counter LSBs:** This part consumes most of the space in each Shadow Table entry and contains part of the LSBs of counters in that Merkle Tree node. 8 LSBs from each counter of the MT node is packed together into 49 bytes(49 bit each).

Whenever 49-bit LSBs of a counter overflows, the MT node is persisted so that the LSB value stored inside SMT can successfully recover the counter value. This ensures that the tree counters are recoverable using the MSB of the memory version of the counters, and the LSBs in the shadow block. Since 49-bit LSB overflows very rarely, the overhead of persistence due to overflows is negligible.

Protecting Shadow Table: As mentioned earlier, since, the original root can be stale and hence no longer can be used for verifying integrity, a small non-parallelizable Merkle tree structure is maintained just to provide integrity protection of the Shadow Table (ST). For 256kB Cache size, only a tree of four levels (8-ary) needs to be maintained. However, there is no need for persisting this tree in memory. It is sufficient to securely keep the root of such a tree, we call it (SHADOW_TREE_ROOT), as verification is done only during recovery and is very fast. It should be noted that, in AGIT scheme, such secondary tree to protect the shadow table is not necessary; if attacker omits or tampers with entries in shadow caches, then the resulting corruption in counters or Merkle Tree will be eventually detected due to root mismatch. To avoid potential deadlock scenario when evictions could occur due to insertion of blocks from the shadow region tree, we dedicate a small percentage of the metadata cache for the shadow region tree. Such part of the cache does not need to

be shadowed. Note that, only keeping higher level nodes of the shadow tree in the cache is sufficient for a fast update of the SHADOW_TREE_ROOT. To protect a 256kb metadata cache, a three-level tree (8-ary, excluding root) is required. We only keep the highest two levels (72 nodes) which is only 1.75% of the cache, and does not affect the performance. The atomicity of the updates to the shadow region in memory and SHADOW_TREE_ROOT is maintained in the same way the atomicity between data, counters and Merkle-tree root is maintained using persistent registers and Write Pending Queue (Section 2.6), in which the root value is only updated once the updates are pushed to the WPQ.

3.3.2 ASIT Recovery Process.

Algorithm 2 ASIT Recover Algorithm

```
1: Read ST
2: Regenerate SHADOW_TREE_ROOT and verify
3: Recover Tree Nodes
4: for STi in ST do
5:     Stale_Nodei ←Read node at address(STi) and place in cache
6:     Recoverd_Nodei ←Replace LSBs and MAC in Stale_Nodei from STi
7: end for
8: Verify Integrity
9: for all Recovered_Nodei in Metadata_Cache do
10:    Verify_Integrity(Recovered_Nodei)
11:    IF Integrity_Not_Verified(Recovered_Nodei) THEN
12:    The system is unrecoverable
13:    ELSE
14:    Do nothing
15: end for
```

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.The following steps are required for the recovery in the ASIT scheme. First, Anubis reads the Shadow Table (ST) from the memory into the cache and regenerates SHADOW_TREE_ROOT, the root of the general tree that is responsible for the integrity of the Shadow Table. Next, this root is compared with the securely stored version of it in the on-chip NVM register. Later, once the ST's integrity has been verified, recovery starts by iterating over each Shadow Table block that has been loaded in the cache. For each Shadow Table block, their non-persisted memory counterpart (stale node) is also read and the LSBs and MAC values of that non-persisted node are replaced with the LSBs and MAC stored in the Shadow Table, i.e., only MSBs of counters are used from the stale node. Later, for each recovered node, we verify that MSBs were not tampered with by verifying the MAC value with the result of applying hash over the counters of the node and the counter in the upper node (from the cache if it was recovered).Once the recovery is done, every recovered tree node will have the dirty bit set to 1. This way, the updates lazily propagate to the memory due to natural eviction.

4 Phoenix Design

4.1 Phoenix VS Anubis

The goal of Phoenix is to enable recovery of Tree of Counters (ToC) integrity trees (SGX-style) at a low write overhead. We realize that any practical solution proposed for NVMs must have low write overhead. Thus, Phoenix mainly aims for ultra-low write overhead while still enabling recovery of ToC integrity trees. The first observation that Phoenix builds upon is that recovery of ToC integrity trees can be achieved by recovering the lost content of security metadata cache. While this observation has also been made in Anubis [2], enabling such a recovery of cache content has been done in a way similar to write-through, by persisting the writes made to security metadata cache into a shadow region in the NVM, which has been proven to be very expensive when used with NVMs [26]. However, Phoenix is based on the fact that we can actually recover the cache content without exact/accurate shadowing of all of its content. We make a novel observation and contribution that we can securely recover the lost cache contents and verify them while still relaxing the shadowing operation.

In particular, we observe that recovering ToC integrity trees by relying on restoring the cache content before a crash has two major requirements. First, there must be a mechanism to verify that we recovered the most recent cache content before a crash and its contents have not been tampered with. Second, the root of the Merkle Tree must reflect the updates of all memory, including the cache contents just before a crash. By ensuring these two requirements are satisfied, the security metadata cache can be recovered, and the rest of the memory verification is verified through a Merkle Tree on each memory access. In other words, simply bringing the metadata cache and Merkle Tree root (unaffected) to the state before a crash is sufficient to ensure the crash-consistency of security metadata.

Anubis, achieved such a cache recovery mechanism by relying on the shadow cache, in which any updates of a lazy-update ToC metadata cache is copied, thus resulting in doubling the writes. To ensure the integrity of the shadow cache, Anubis applies a small Merkle Tree over the shadow cache while keeping its root in the processor and following an eager update scheme. After a crash, the cache content can be restored from the shadow region, and its integrity can be verified using the small Merkle Tree (the eagerly updated one), which also has its root kept in an NVM (or NVM-backed) register inside the processor chip. On the other hand, Phoenix is mainly based on the fact that most updates to metadata cache in the lazy-update scheme are for leaves. However, shadowing leaves updates to the memory might be unnecessary if we can have the following: ① a mechanism to verify the most-recent cache state including leaves but without necessarily shadowing them, and ② the ability to recover leave updates.

Phoenix employs Osiris and phase-based recovery [26], to relax updates to the shadow region in the cache while simultaneously allowing to recover the exact content of the cache right before a crash. Specifically, Phoenix selectively decides which security metadata should be shadowed strictly and which ones can be relaxed. Even though it relaxes the shadow region update, Phoenix enables the reconstruction of the cache content (including relaxed leaves) and allows the verification of the recovered content. Since most updates to the security metadata cache are caused by leaves updates, Phoenix is expected to significantly reduce the number of writes while allowing fast recovery of ToC trees. The main

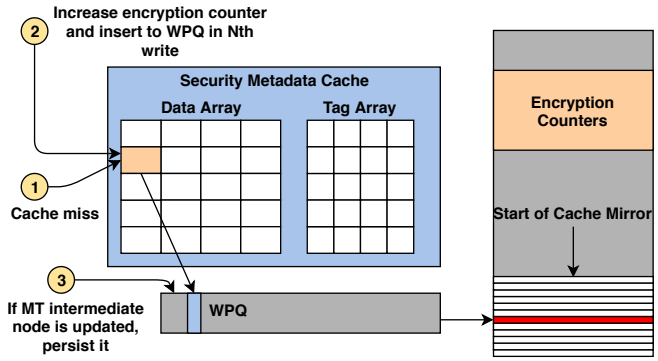

Fig. 10. Updates Tracking

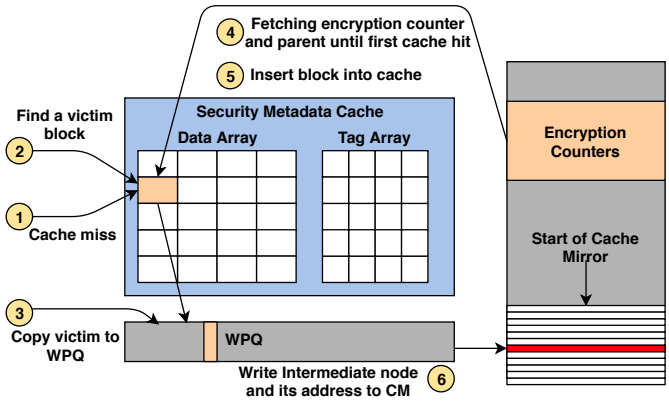

Fig. 11. Eviction

downside of Phoenix is that it requires additional work before reconstructing the lost cache content and verifying it.

4.2 Selective Persistence

Upon a crash, the cache loses its content. Losing the cached security metadata results in integrity verification failure; thus, the cached security metadata needs to be persisted.

Strictly persisting the security metadata incurs tens of additional writes, and to avoid those unnecessary extra writes, we opt for persisting only the unrecoverable nodes of the metadata. Security metadata contains the encryption counters and the Merkle tree nodes, while ToC integrity tree nodes are composed of 8 counters and a MAC calculated over the eight counters and the parent of the node. Since the encryption counters can be retrieved without strictly persisting them using Osiris [26], we are going to follow a similar scheme by persisting the encryption counters with every N-th write, or on eviction. On the other hand, the intermediate ToC nodes are not recoverable; therefore, we suggest persisting these cached nodes to successfully recover from the crash. To achieve that, we allocate a small region in the NVM which is the same size of the security metadata cache (about 256 kB) which we refer to as the Cache Mirror (CM). Whenever an intermediate ToC node is written in the cache, this update will be persisted, and its address will be copied to the CM, while updating the encryption counters with every N-th write. Figure 10 shows how selective persistence is done; whenever a write happens, if the write resulted in updating an intermediate node, the updated intermediate node is copied to the CM region. During the recovery process, the contents of the CM are used to recover the lost cache contents and refresh the ToC to ensure a secure recovery process. Since the security region is defined by the boundaries of the processor, the integrity of the CM should be guaranteed before it can be used during the recovery. Thus, we apply a small Merkle Tree (MT), four levels with an arity of eight, over the CM while keeping the root of this tree in the processor. During the recovery, the integrity of the CM region is verified by building the CM-MT and comparing the resulting root with the processor kept root.

4.3 Phoenix Operation

Phoenix read operation is merely a read and verify operation, and does not require any changes or special handling. In particular, the read operation in Phoenix does not modify the security metadata cache except for eviction, which is discussed in subsection 4.5. On the other hand, the write operation results in an encryption counter increment to ensure a new encryption pad for the modified block. The encryption counter increment will not affect the MAC value in the node nor increment the parent as we are using a lazy update scheme, but Phoenix will be triggered and the address of the modified counter will be copied to the CM. However, when an encryption counter block is evicted the parent node should be fetched and both nodes should be updated. Despite using a lazy update scheme, it is important to persist the encryption counter at every N-th write, or on eviction to enable encryption counter recovery. It is important to keep in mind that updates of a ToC node and the data are to be done atomically using the Write Pending Queue (WPQ) and a ready bit as described earlier. While encryption counters are updated at every N-th write, ToC intermediate nodes need to be persisted each time they are modified; thus, the addresses of the intermediate nodes are copied to the CM, and the intermediate nodes are persisted into the NVM.

4.4 Phoenix$^+$ Operation

While Phoenix persists intermediate nodes on each update, and persists encryption counters on N-th update or eviction. Phoenix$^+$ relaxes persisting encryption counters on eviction, to only persist encryption counters on the N-th write. By doing this, Phoenix$^+$ reduces the number of writes and the performance overhead significantly. Phoenix$^+$ relies on recovering the encryption counters while working by utilizing the encryption counters recovery scheme. Notice that, recovering the encryption counters on the run might add performance overhead if done in a sequential manner, but we assume N-AES engines (4 in our design) to retrieve the latest value of the used encryption counter. Keep in mind, that evicting an encryption counter without updating its value, does not affect its parent, but still affect the encrypted data. Thus, the old encryption counter value integrity can be verified the parent value, and the latest value can be recovered.

4.5 Eviction

The lazy update scheme we use in Phoenix reduces the number of writes while relying on eviction to propagate the nodes update. Figure 11 shows the eviction process. In the case of a counter encryption cache miss, the memory controller selects a victim block to be evicted from the cache using the Least Recently Used (LRU) replacement policy. The victim block is then inserted to the WPQ in case it was an intermediate tree node. Note that we are not

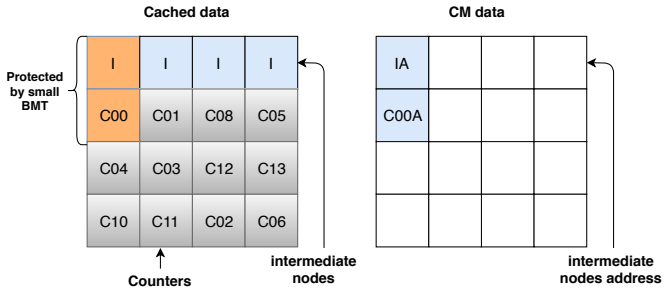

Fig. 12. Cache Mirror

persisting the encryption counter on eviction since we are relying on Osiris to retrieve the encryption counter's most recent value while running. Persisting the encryption counter on eviction will improve the performance slightly, as we can immediately use the fetched encryption counter after verifying its integrity, although will increase the number of writes to the NVM. Since our goal is to successfully recover the ToC while maintaining a low number of writes, we opt for recovering the encryption counter while running by only persisting the encryption counter at every N-th write. To ensure the data consistency, we assume the evicted encryption counter, intermediate ToC node, data, and CM data are inserted atomically to the WPQ as described in section 2.6.

4.6 Imprecise Cache Mirror

The Cache Mirror (CM) region, shown in Figure 12, is a small reserved region in the NVM. The CM only contains the addresses of the dirty intermediate nodes and the addresses of the dirty counters, while the actual dirty intermediate nodes are persisted to their actual locations. The CM contents are used to securely recover the system after a crash. To ensure the integrity of recovery, the dirty cached intermediate nodes and the dirty encryption counters are protected with a small general MT. This small MT that only covers the dirty intermediate nodes and dirty encryption counters is eagerly updated, and its root is always kept in the processor. Notice that by relaxing the CM contents to contain only the addresses of dirty intermediate nodes and the addresses of dirty encryption counters, we were able to drop the number of writes significantly. Moreover, using a small MT to cover only the dirty intermediate nodes and dirty encryption counters, we were able to drop the performance overhead. We note that when a lazy update scheme is used, the root is no longer suitable as a single point of memory content integrity verification. As a matter of fact, the cache contents are the most updated nodes, and the nodes are used to verify the integrity of fetched nodes. When a crash happens, and the cached nodes are lost. However, we can recover the leaf nodes using encryption counters recovery, although the integrity of these nodes needs to be verified. The parents of these nodes can be either up-to-date in the NVM, or cached nodes lost during the crash. Thus, we make sure to persist the intermediate nodes, and use the small MT root to ensure the integrity of cached intermediate nodes.

4.7 Integrity Verification

The secure region is defined by the processors boundary. While on-chip memory is considered secure, that is not the case with NVM. Thus, whenever a data block is fetched from the NVM its integrity needs to be verified. To verify the integrity of any block, its parent needs to be fetched and used to calculate the MAC value of the verified block. However, once the parent is fetched, its integrity needs to be verified which will result in a recursive operation until the first parent cache hit. Once one parent is found in the cache, its integrity is considered to be verified, and is used to calculate the MAC of the child node. If the calculated MAC value matches the child node's stored MAC value, the child's integrity is considered verified. For the CM region, since its size is very limited (256 kB) it is more suitable to use an eagerly updated MT and store its root in the processor. Using an eagerly updating scheme means the root always reflects the most recent tree state. The CM MT is four levels using an 8-ary tree; thus, it is feasible to rebuild the tree during recovery and compare the new root with the stored root to verify its integrity.

4.8 Recovery

The recovery process starts by loading the pre-crash cached intermediate nodes from the NVM, using the addresses saved in the CM region. Then, the integrity of the loaded intermediate nodes is verified using the small MT root. When the intermediate nodes are verified, any interrupted write operation is resumed, by checking the DONE_BIT and completing the pending operations to successfully complete the atomic write. Notice that the small MT root is eagerly updated, and always kept in the processor. Moreover, the small MT root is calculated over the dirty cached intermediate nodes; thus, its update is infrequent, since most of the updates are done to the leaf nodes. In turn, the overhead of eagerly updating the small MT root is negligible. Note that we are restoring the encryption counter during the normal operation, and the encryption counters are not persisted nor recovered during the recovery process since they are recovered when fetched.

Algorithm 3 Phoenix Recover Algorithm

```
1: Read CM
2: for Nodei in CM do
3:     Nodei ←Memory[Nodei]
4:     if Nodei is EncryptionCounter then
5:         Nodei ←Osiris[Nodei]
6:     end if
7:     MetadataCache ←Nodei
8: end for
9: CMRoot_Recovered ←GenerateRoot[MetadataCache]
10: if CMRoot_Recovered = CMRoot_Stored then
11:     WPQ ←PersistentRegisters
12:     Memory ←WPQ
13:     ContinueNormalOperation
14: else
15:     TheSystemisUnrecoverable
16: end if
```

Once the CM integrity is verified, the DONE_BIT is checked and any pending write operations that were in the persistent registers before the crash are moved to the WPQ and executed. After the pending write operations are executed, the CM contents are used to restore the cached intermediate ToC nodes. While the intermediate ToC nodes are ensured to be recovered to the most recent state using the CM, the encryption counters are not. To restore the encryption counter to the most recent state, we use the CM contents to retrieve the addresses of the cached encryption

counters, then fetch the counters and use Osiris to retrieve the most recent counter value. After the encryption counters are updated to the most recent values, the cache is restored to its previous state before the crash, and its integrity can be verified using the small MT root. Notice that in case of the CM region is tampered with, and the calculated root of the CM region does not match the stored root in the processor, the recovery process fails, and the integrity of the NVM is declared unverifiable.

4.9 Phoenix VS cc-NVM

Phoenix and cc-NVM [25] aim to achieve a persistently secure and crash-consistent system. However, Phoenix and cc-NVM are following different approaches to achieve a common goal. While Phoenix focuses on recovering the ToC with ultra-low writes overhead, cc-NVM enables the crash consistency for BMT and does not support the ToC. cc-NVM enables the recovery by tracking the security metadata updates in a dirty-address-queue and persists the updates by the end of each epoch. cc-NVM assumes the tree nodes are recoverable by hashing the encryption counters, which is not the case for ToC. Moreover, cc-NVM epoch persistency increases the write traffic to the NVM. On the other hand, Phoenix persists the updated intermediate ToC nodes and recovers the encryption counters. Therefore, Phoenix reduces both the performance overhead and write traffic.

4.10 Security Discussion

In traditional persistent secure systems, the security of the data is protected using the counter mode encryption, and the integrity of the encryption counters is protected with MT. The root of the MT is always kept in the processor, and memory content's integrity is verified by calculating the root and comparing it with the processor stored root. This scheme works well for eagerly updated MT, which is not the case in our scheme. Phoenix^{+} scheme relies on a lazy update scheme, which means whenever a leaf counter is updated or evicted we do not update the parent of the counter, nor update the associated MAC with the leaf counter node, but rely on the N-th write to the same counter to propagate the update. In Figure 3, if the counter `C01` is updated twice and then got evicted, the parent node `B00` will not be updated, and the MAC value `MAC00` will both be stale. Notice that even the counter `B00` will be stale in the NVM, so the next time counter `C01` is fetched it still can be verified successfully using the stale `MAC00` and the parent `B00`, and then its most recent value can be recovered using Osiris [26]. In the lazy update scheme, the root of the ToC can be stale, and the most updated state is preserved in the cached intermediate nodes of the ToC. In the recovery process, it is essential to guarantee the integrity of the CM region as it reflects the most recent state of the tree. Thus, the integrity of the CM is protected using a small BMT and the root is eagerly updated and kept in the secure region (processor).

5 EVALUATION

In this section, we evaluate our scheme based on the `Write Back scheme` as the baseline. We evaluate the additional number of write incurred by each scheme, the performance of ASIT, Phoenix and Phoenix^{+} schemes, then we show the sensitivity to cache size and recovery time.

TABLE 1
Configuration of the simulated system

Processor	
CPU	4 Cores, X86-64, Out-of-Order, 1.00GHz
L1 Cache	Private, 2 Cycles, 32KB,2-Way
L2 Cache	Private, 20 Cycles, 512KB, 8-Way
L3 Cache	Shared, 32 Cycles, 8MB, 64-Way
Cacheline Size	64Byte
DDR-based PCM Main Memory	
Capacity	16GB
PCM Latencies	Read 60ns, Write 150ns
Encryption Parameters	
Security Metadata Cache	256KB, 8-Way, 64B Block
CM in Phoenix	256KB
CM in Phoenix^{+}	256KB
Persistence Limit	4

5.1 Methodology

Our evaluation was done using Gem5 [7], a cycle-level simulator. Table 1 shows the used configuration, we simulate a 4-core X86 processor with 16GB PCM-based Main Memory with parameters modeled as in [15]. The evaluation was done by running applications from the SPEC 2006 benchmark suit [12]. The used benchmarks include memory intensive applications in both read and write intensive applications. For each application, we simulate 500M instructions after fast forwarding to a representative region.

In our evaluation, we model the integrity-protection using ToC, encryption aspects, security metadata cache, hash calculation latency, shadow tables, and cache mirror region integrity protection.

5.2 Phoenix Writes

To evaluate our scheme, we compared the number of writes incurred for each of the following schemes.

1) **Write Back (Baseline)**: This is a simple ToC integrity tree scheme with write back. This system only writes on eviction and does not provide any recoverability.
2) **ASIT**: The Anubis scheme for ToC integrity tree updates the ToC lazily and writes all the ToC updates to a shadow region.
3) **Phoenix**: This scheme updates the ToC lazily while persisting the updates for intermediate ToC nodes, and relaxes the updates for leaf nodes until eviction or the counter is written N times.
4) **Phoenix^{+}**: This scheme reduces the number of writes in Phoenix by only persisting the leaf nodes on the Nth write, and relies on a counter recovery scheme (Osiris [26]) to recover the counters on the run.

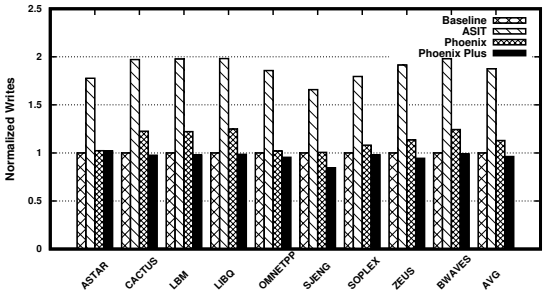

Fig. 13. Phoenix Extra Writes

Figure 13 shows the number of writes incurred by the above schemes. Considering the `Write-Back` as the baseline scheme, we notice that Anubis-ASIT incurs an average of 87% extra

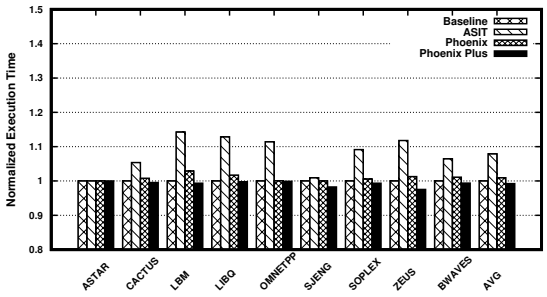

Fig. 14. Phoenix Performance

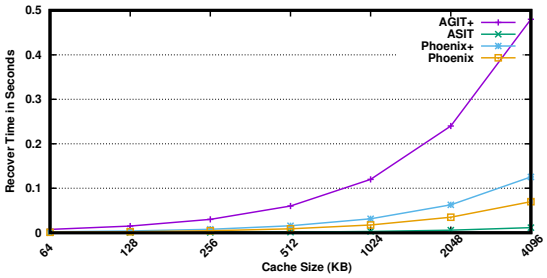

Fig. 15. Phoenix recovery Time

writes, while Phoenix incurs an average of 12.9% extra writes, and Phoenix$^+$ reduces the writes to less than the write-back by an average of 3.8%. Phoenix$^+$ reduces the number of writes to less than Write Back scheme while achieving the recoverability of ToC. Anubis-ASIT overhead is caused by the strict shadowing of the ToC updates, on the other hand, Phoenix only persists when intermediate nodes are changed, or the dirty encryption counter is evicted, and Phoenix$^+$ only persists on encryption counter fourth update and relax the persist for evictions. Phoenix$^+$ achieves this reduction by utilizing the lazy update scheme for the ToC, and by eliminating the eviction writes for the encryption counter nodes, while using Osiris counter recovery to recover the latest value of the encryption counter each time it is fetched.

5.3 Phoenix Performance

To evaluate Phoenix, we model and compare the aforementioned four schemes. Figure 14 illustrates Phoenix's performance in comparison to other schemes. Considering the Write Back scheme as the baseline, Anubis-ASIT provides the ability to recover the ToC with 7.9% extra performance overhead. Phoenix$^+$ is not only capable of recovering the ToC, but also achieves a performance of 0.8% better than the write back scheme. That is, Phoenix$^+$' execution time is less than the Write Back scheme; thus, Phoenix$^+$ reduces the overhead by 8.7% compared to Anubis-ASIT. Note that ASIT incurs higher overheads due to the high number of writes, which leads to contesting the shared resources such as the WPQ. On the other hand, Phoenix reduces the tracking to the first change only, and thus improves the system performance. We also notice that Phoenix in both versions is performing better than the baseline for *CACTUS* benchmark as shown in Figure 14. Moreover, using memory intensive benchmarks shows that Phoenix$^+$ performs slightly better than the Write Back scheme, while this difference is expected to be more noticeable with less memory intensive applications. Phoenix reduces the overhead by relying on lazily updating the ToC while persisting each update to the intermediate ToC nodes. Notice that relying on the lazy update reduces the frequency of updating the intermediate nodes until the leaf node is evicted. On the other hand, Phoenix$^+$ takes one step further to relax persisting the leaf counters on eviction and relies on Osiris as a counter recovery scheme to recover the counters while running.

5.4 Phoenix VS Phoenix$^+$

Phoenix$^+$ is a modified version of Phoenix that reduces the writes and improves the performance. However, Phoenix$^+$ achieves these performance gains and writes reduction by increasing the hardware complexity. Phoenix$^+$ requires multiple AES engines to recover the encryption counters online. The number of AES engines should be equal to used encryption counters persistence limit to achieve optimal performance, as using less number of engines will lead to a sequential encryption counter recovery process, which might affect the system performance. Therefore, Phoenix and Phoenix$^+$ solve the same problem where Phoenix optimizes for lower hardware complexity and lower recovery time but incurs extra writes and performance overhead, and Phoenix$^+$ optimizes for better performance and lower writes at the cost of hardware complexity and extra recovery time.

5.5 Sensitivity Study

5.5.1 Recovery Time

Recovery of ToC protected systems was not possible except for strict persisting scheme. Anubis-ASIT [2], Phoenix, and Phoenix$^+$ allow the recovery in less than a second, due to making the recovery time a function of the cache size instead of the memory size. While Anubis-ASIT relies on a lazy strict persistent scheme which results in extra 87% extra writes to achieve the recoverability of the ToC integrity protected systems, on the other hand Phoenix$^+$ recovers the same NVM with writes less than `Write-Back` scheme. Figure 15 shows the recovery time of Anubis, and Phoenix regarding the cache size. Figure 15 shows that both schemes achieve a recovery time of less than a second, even for extremely large cache size (4MB) Phoenix recovery time is ≈0.12 seconds for Phoenix$^+$, 0.069 for Phoenix, and ≈0.015 seconds for Anubis-ASIT. Notice that Phoenix$^+$ recovery time is the highest as it needs to recover many nodes, and Anubis-ASIT scheme has the lowest as it does not need to recover any node, and Phoenix recovery time is between the two. Phoenix$^+$ takes more time to recover the system as it requires retrieving the leaf counters values during the recovery process. For this sensitivity test, the worst case scenario is considered to calculate the recovery time, by considering all the eight counters in each leaf node are stale. We notice that Phoenix$^+$ trades a very small amount of recovery time for reducing performance overhead and the number of writes of the system.

5.5.2 Performance Sensitivity to Cache Size

Phoenix$^+$ allows the recovery of ToC integrity protected NVM as a function of the cache size. To fully evaluate the scheme, we vary the cache size and measure the performance overhead of our scheme. Figure 16 shows that the performance of Anubis-ASIT and Phoenix$^+$ almost stay the same, while Anubis-ASIT incurs an overhead of 8%, Phoenix$^+$ performance is about 1% better than `Write-Back` scheme. This can be explained by Phoenix

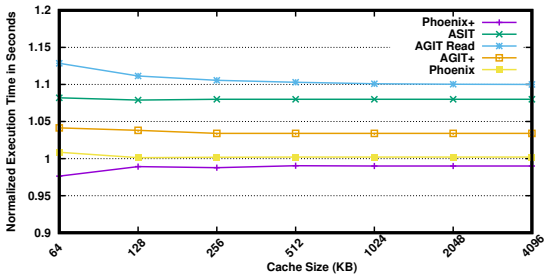

Fig. 16. Phoenix sensitivity to Cache Size

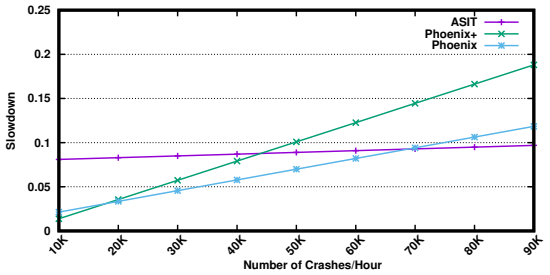

Fig. 17. Performance Sensitivity to Crashes

operation; Phoenix$^+$ performs in a manner similar to the baseline (Write Back). However, Phoenix$^+$' performance depends on the number of writes to cached data: the more writes to the cached data will result in more counter writes, which results in more Phoenix writes.

5.5.3 Performance Sensitivity to Crashes

Figure 17 illustrates an analytical comparison between Phoenix, Phoenix$^+$ and ASIT. It shows how the normalized system slowdown varies with the number of crashes per hour. The slowdown in our calculation includes slowdown during system runtime, and slowdown during recovery process. Because of the straight forward recovery mechanism in ASIT, it achieves low recovery time at the cost of significant run-time performance implications. On the contrary, Phoenix and Phoenix$^+$ have improved performance at the cost of slightly higher recovery time. Therefore, Phoenix and Phoenix$^+$ have better performance if the frequency of crashes is small. However, with a higher frequency of system crashes, the performance gains of Phoenix and Phoenix$^+$ are obliterated by the extra time needed for the recovery.

As shown in Figure 17, Phoenix$^+$ shows the best performance for systems with a crash rate lower than 17K per hour. Moreover, We observe that ASIT scheme shows better performance than Phoenix$^+$ if the crash rate is higher than 44K per hour. However, Phoenix would still perform better than ASIT for systems with a crash rate that is lower than 70K per hour. Therefore, for ASIT can perform better than Phoenix$^+$ in systems that crashes more than 12 times each second. Although such a high frequency of system crash is uncommon, we perform such sensitivity analysis to have an insight on how the two schemes would perform in a system that is vulnerable to frequent system crash. In most servers where a system crash is rare, Phoenix and Phoenix$^+$ can be used. However, in systems that are susceptible to frequent crashes and power loss (e.g., intermittent power devices) ASIT can be used.

5.5.4 Counter Persistence Limit

The number of writes on which the encryption counter is persisted affects the performance of Phoenix. Using a large number of writes before the encryption counter is persisted reduces the number of writes and the performance overhead. However, this comes at a cost: a large persistency limit would cause higher recovery time and higher performance overhead as the counter latest value needs to be recovered each time its fetched or during recovery. The performance overhead can be avoided by using multiple ECC engines to recover the counter value. In our design, we opt for using the 4th write to be the persistence limit, choosing to persist the counter at the 4th write provides a very low performance overhead and enables the recovery within less than a second.

6 RELATED WORK

The most related work to Phoenix are Anubis [2], cc-NVM [25], Triad-NVM [6], Osiris [26], and Crash Consistency [18]. Anubis [2] addresses the recovery time of NVM systems and uses a shadow region to track down all the changes to cache contents, where each writes to the cache results in a write to the shadow region. The shadow region facilitates recovering the cache contents in ultra-low time, but incurs 87% extra writes for ToC. cc-NVM [25], proposes a crash consistency scheme that uses an epoch-based persistency model. cc-NVM tracks the updated BMT nodes in a dirty-address-queue and commits the updates once the queue is full. Thus, cc-NVM reduces the traffic to the NVM and improves the system performance. Triad-NVM [6], on the other hand, discusses the trade-off between recovery time and performance, and reduces the recovery time by persisting N levels of the MT. On the downside, Triad-NVM does not work with ToC, and requires persisting multiple levels of the integrity tree.

To recover counters, Osiris [26] relies on the ECC-bits as a sanity check for the used encryption counter. By applying a stop-loss mechanism, Osiris restores the encryption counters using a limited number of trials. Osiris works for retrieving the encryption counters while assuming building the integrity tree is possible, which is not the case with ToC integrity tree. Crash Consistency [18] for counters recovery proposes an API for programmers to selectively persist counters, and ensures atomicity through a write queue and Ready-Bit. In order to reduce the overhead, it proposes selective counter atomicity of the persistent applications. The scheme depends on the amount of applications persistent data and does not address the recoverability issue of ToC.

There are several state-of-the-art works done in NVM security and persistence [9], [14], [19], [21], [22], [23], [28], [29] without considering the crash-consistency and recovery that discusses to optimize the run-time overhead of implementing security to NVM. Most works employ counter-mode encryption for encrypting data and MT for ensuring integrity. However, to the best of our knowledge, none of the works consider the recovery and crash-consistency of integrity protected systems. As a matter of fact, any work that does encryption counters compression or increases the integrity tree arity boosts our scheme, by increasing the cacheability of the encryption counters and reducing the number of intermediate nodes. SecPM [29] proposes a write-through mechanism for the counter cache that tries to combine multiple updates of counters to a single write to memory, however, does not ensure recovery for ToC and incurs significant recovery time as in Osiris. While Anubis [2] discusses the reduction of recovery time of secure non-volatile memory and recovery mechanism for

ToC, however; the scheme incurs almost 2x extra writes which reduces the NVM lifetime by half.

7 Conclusion

Our work bridges the gap between recoverability and high performance in secure Non-Volatile Memories. Our solution can be seamlessly integrated into various secure and integrity protected systems. Anubis scheme can achieve significant improvement in recovery time and incurs an overhead of 3.4% when implemented in general Merkle Tree integrity-protected systems. Phoenix is based on four observations, first, most updates of the lazily updated ToC are done to leaf nodes. Second, leaf nodes are the least likely to be evicted as they will be reused frequently for verification and update purposes. Third, leaf nodes can be recovered using any encryption counter recovery scheme, we used Osiris in our work, but any other scheme should work. Fourth, cached intermediate nodes can be persisted at their location instead of being copied to the shadow region, and the small MT only needs to cover the dirty cached intermediate nodes and the dirty encryption counters. Phoenix achieves recoverability with ultra-low recovery time while keeping the number of writes to the minimum in ToC integrity protected NVMs. Our solution achieves a significant improvement in the number of writes as it reduces the number of writes to 3.8% less than the write back scheme, with a recovery time of less than a second in ToC integrity protected systems. In addition, Phoenix recovery time and extra writes are a function of the cache size, as it works by recovering the lost cached ToC nodes. In summary, Phoenix recovers the ToC in less than a second, reduces the number of writes significantly, and improves the performance.

Acknowledgement

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Approved for public release. Distribution is unlimited.

References

[1] "NVDIMM Block Window Driver Writer's Guide," https://pmem.io/documents/NVDIMM_DriverWritersGuide-July-2016_wChanges.pdf, accessed: 2019-12-30.

[2] 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.

[3] A. Awad, P. Manadhata, S. Haber, Y. Solihin, and W. Horne, "Silent shredder: Zero-cost shredding for secure non-volatile main memory controllers," *ACM SIGOPS Operating Systems Review*, vol. 50, no. 2, pp. 263–276, 2016.

[4] A. Awad, S. Suboh, M. Ye, K. Abu Zubair, and M. Al-Wadi, "Persistently-secure processors: Challenges and opportunities for securing non-volatile memories," in *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2019, pp. 610–614.

[5] A. Awad, Y. Wang, D. Shands, and Y. Solihin, "Obfusmem: A low-overhead access obfuscation for trusted memories," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 107–119, 2017.

[6] A. Awad, M. Ye, Y. Solihin, L. Njilla, and K. A. Zubair, "Triad-nvm: Persistency for integrity-protected and encrypted non-volatile memories," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 104–115. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322250

[7] N. Binkert, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, D. A. Wood, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, and T. Krishna, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, p. 1, aug 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[8] S. Chhabra and Y. Solihin, "i-nvmm: a secure non-volatile main memory system with incremental encryption," in *2011 38th Annual international symposium on computer architecture (ISCA)*. IEEE, 2011, pp. 177–188.

[9] 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. 47, no. 4, pp. 105–118, 2012.

[10] V. Costan and S. Devadas, "Intel sgx explained." *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.

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

[12] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, sep 2006. [Online]. Available: https://doi.org/10.1145/1186736.1186737

[13] https://www.forbes.com/sites/kellyclay/2013/08/19/amazon-com-goes-down-loses-66240-per minute/#68dc6b6b495c, "Amazon.com goes down, loses $66,240 per minute." 2019.

[14] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language-level persistency," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 481–493.

[15] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 2–13, 2009.

[16] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *IEEE Micro*, vol. 30, no. 1, pp. 143–143, Jan. 2010. [Online]. Available: http://dx.doi.org/10.1109/MM.2010.24

[17] Z. Li, R. Zhou, and T. Li, "Exploring high-performance and energy proportional interface for phase change memory systems," in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, ser. HPCA '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 210–221. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2013.6522320

[18] 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)*. IEEE, 2018, pp. 310–323.

[19] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu, "Thynvm: Enabling software-transparent crash consistency in persistent memory systems," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2015, pp. 672–685.

[20] 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 *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007, pp. 183–196.

[21] 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.

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

[23] M. Taassori, A. Shafiee, and R. Balasubramonian, "Vault: Reducing paging overheads in sgx with efficient integrity verification structures," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 665–678.

[24] 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, pp. 179–190.

[25] F. Yang, Y. Lu, Y. Chen, H. Mao, and J. Shu, "No compromises: Secure nvm with crash consistency, write-efficiency and high-performance," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[26] 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.

[27] V. Young, P. J. Nair, and M. K. Qureshi, "Deuce: Write-efficient encryption for non-volatile memories," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 33–44, 2015.

[28] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2013, pp. 421–432.

[29] 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)*, 2018.

Mazen Alwadi is a second-year Ph.D. student majoring in Computer Engineering at the University of Central Florida. His research interest is in hardware security, secure computer architecture, security of heterogeneous and fabric attached memory systems. He received holds a bachelor degree in Computer Engineering from the Jordan University of Science and Technology, A Master degree in Computer Engineering from Yarmouk University in Jordan.

Kazi Abu Zubair is a third-year Ph.D. student majoring in Electrical Engineering at North Carolina State University (NCSU). 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 the Ph.D. program.

David Mohaisen is an Associate Professor at the University of Central Florida (UCF). His research interests are in the area of computer systems security. He holds 10 U.S. patents, has published more than 100 peer-reviewed research papers, is serving on the editorial board of IEEE Transactions on Mobile Computing and IEEE Transactions of Parallel and Distributed Systems, among others, and and is a senior member of both ACM and IEEE. Currently, he leads the Security and Analytics Lab (SEAL) at UCF.

Amro Awad is an Assistant Professor at North Carolina State University (NCSU). 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 NCSU.