Root Crash Consistency of SGX-style Integrity Trees in Secure Non-Volatile Memory Systems

Jianming Huang and Yu Hua✉
Wuhan National Laboratory for Optoelectronics
School of Computer, Huazhong University of Science and Technology
✉*Corresponding Author: Yu Hua (csyhua@hust.edu.cn)*

***Abstract*—Data integrity is important for non-volatile memory (NVM) systems that maintain data even without power. The data integrity in NVM may be compromised by integrity attacks, which can be defended against by integrity verification via integrity trees. After NVM system failures and reboots, the integrity tree root is responsible for providing a trusted execution environment. However, updating the root incurs latency to propagate the modifications from leaf nodes to the root. If system crashes occur during the propagation process, the root is inconsistent with the updated leaf nodes, resulting in misreporting the attacks after the system reboots. In this paper, we propose a ShortCut UpdatE scheme, called SCUE, which is an efficient and low-latency scheme to instantaneously update the root on the SGX-style integrity tree (SIT) by judiciously overlooking the updates upon most intermediate tree nodes. The idea behind SCUE explores and exploits the observation that consistent leaf nodes and root are enough to ensure data integrity after system failures and reboots. Moreover, SIT is difficult to be reconstructed from the leaf nodes since updating one tree node needs its parent node as input. Root in SIT thus cannot verify the data after the system crashes and reboots even though the root is correctly updated. To provide the ability of verification via root in SIT, we use a counter-summing approach to efficiently reconstructing the SIT from leaf nodes. Extensive evaluation results show that compared with the state-of-the-art integrity tree update schemes, our SCUE delivers high performance while ensuring data integrity.**

I. Introduction

Non-Volatile Memory (NVM) has demonstrated the salient features of non-volatility, low power consumption and high performance. To guarantee the data security and confidentiality, we need to encrypt the data in NVM. Moreover, the data integrity is also important, which is interpreted that the data cannot be illegally tampered [44]. The encrypted data need to be further verified to ensure the integrity, which requires security metadata, i.e., counter blocks in counter mode encryption [56] and tree nodes in integrity tree verification [23]. Due to the persistence of NVM, these security metadata need to be crash consistent and recoverable, to guarantee that the NVM systems continue to run safely and efficiently after system failures and reboots [60].

To ensure data confidentiality, existing designs use the counter mode encryption (CME) [56] to encrypt the data in NVM. CME uses counters to generate the one-time paddings (OTPs) and XORs the OTPs with the plaintext/encrypted data to generate the encrypted/plaintext data. The generation of OTPs is in parallel with reading the data from NVM. Therefore, the latency of decryption is hidden by that of reading data. The data integrity is generally verified by the integrity trees, such as Merkle Tree (MT) [20], [28], [37], Bonsai Merkle Tree (BMT) [7], [19], [41], [44] and SGX-style Integrity Tree (SIT) [4], [14], [31]. In the integrity tree, the user data are iteratively hashed to generate the keyed-Hash Message Authentication Codes (HMACs) in the intermediate tree nodes (in MT/BMT), or connected with the intermediate nodes via the counters and HMACs (in SIT). The changes in the root of the integrity tree reflect the modifications of the user data.

Compared with BMT, SIT demonstrates high performance (updating nodes in parallel) [60], high security (one counter protected by two HMACs) [5], and low storage overheads (more than 8 counters compressed in one node) [45], [50]. However, efficiently using SIT to protect data integrity still suffers from two problems: the recovery failure due to the *root crash inconsistency* (§III-B) and high persistent overheads due to the *complicated dependencies* (§III-D) among SIT nodes.

The root in the integrity tree is the only trusted base to verify the integrity of data [19]. Since the root is stored in the non-volatile register on-chip, the root cannot be attacked in our threat model (§II-A). Unlike the root, other tree nodes need to be flushed into NVM and are possibly tampered by the attackers. The tree nodes in NVM are hence not trusted. When running the system, the cached intermediate tree nodes are regarded as the trusted bases since they are verified by the root. However, when the system failures occur, the cached intermediate nodes are lost or flushed into NVM, and only the on-chip root is trusted. For verifying data after the system reboots, we need to instantaneously update the root during system running.

When the updated leaf nodes are persisted, the modifications of leaf nodes need to be propagated to the root for verification after system failures and reboots. Propagating the modifications needs to iteratively hash the data in the tree nodes. One hash computation has to consume 40 cycles [20], [34], [47]. This becomes exacerbated due to the high integrity tree in large NVM, e.g., tens of levels in the integrity tree for the 64GB NVM. Propagating the modifications thus requires tens of hash computations, which incurs a long latency in the systems. The latency of propagating modifications from leaf nodes to root is called *the root crash inconsistency window*, or *crash window*. Fortunately, SIT can update these HMACs in parallel to significantly reduce the propagating latency [5], [60]. But the *crash window* in SIT still exists. When crashes occur during the *crash window*, the propagation process of the modifications aborts. Therefore, the root stored in the on-chip non-volatile register is not updated. After system recovery, the un-updated root does not match the updated leaf nodes in NVM, causing *the root crash inconsistency problem*. Specifically, the integrity tree is reconstructed from the updated leaf nodes, and a new root is reconstructed. Since the un-updated root stored in the non-volatile register on-chip is not the same as the reconstructed root, the recovery fails even though no attacks occur.

Intel Asynchronous DRAM Refresh (ADR) technique [1] flushes data from write pending queue (WPQ) in memory controller into NVM when the system crashes. Moreover, Intel recently proposed Extended ADR (eADR) [2], [3], [15] to flush the data from CPU caches into NVM after system crashes. By using ADR/eADR, the on-chip caches and WPQ become persistent domains.

The use of eADR is assumed to improve the performance and security of the system [19]. However, in eADR, the root crash inconsistency problem, although mitigated but not addressed, still exists. eADR is used to flush data into NVM after system crashes *without the ability to read and process data from NVM*. However, updating root upon crashes needs to read the intermediate nodes from NVM and compute the HMACs in the propagation path, which is not supported by eADR after system crashes. When crashes occur, the root is not updated and does not match the leaf nodes after a reboot even with eADR.

Root crash consistency has been discussed by PLP [18] and BMF [19]. However, PLP and BMF only focus on BMT due to the complicated dependencies among SIT nodes. In BMT, one node is constructed by hashing its child nodes. But in SIT, updating one node needs its parent node as input, i.e., low-level nodes depend on high-level nodes. Only consistently persisting leaf nodes and updating root are not enough to protect data integrity after system crashes and reboots. SIT cannot be recovered from the leaf nodes. Even if we correctly update the root, the system does not know whether the leaf nodes are attacked or not via checking the stored SIT root [60]. For SIT, PLP and BMF fetch, update and persist all nodes in the branch from leaf node to root to ensure crash consistency with large overheads [18], [19].

We observe that in SIT, increasing the child counter causes the increment of the parent counter. The parent counter hence is the sum of the corresponding child counters. Moreover, the counter in the root is the sum of all corresponding leaf counters. Based on this observation, to address the *root crash inconsistency problem*, we focus on SIT and propose the **S**hort**C**ut **U**pdat**E** scheme (SCUE) to correctly and efficiently update the root of SIT with low overheads. The idea behind SCUE is that only the consistent leaf nodes and root are required for recovery after system crashes. Our SCUE also provides a *counter-summing* approach to decoupling the complicated dependencies among SIT nodes. Thus, SIT can be reconstructed from leaf nodes up like the BMT.

To evaluate the performance of our proposed scheme, we use Gem5 [11] with NVMain [39] to implement SCUE. We evaluate 5 typical persistent workloads, and 8 macro-benchmarks from the SPEC2006 [22] which have been widely used and well-recognized in the community [13], [23], [26], [27], [33], [43], [61]. Our experimental results show that SCUE only incurs 7% performance overhead even if compared with the insecure baseline system.

In summary, this paper makes the following contributions:

- **Shortcut update scheme for root crash consistency.** We propose a new update scheme to instantaneously update the root of the integrity tree without propagating the modifications on the intermediate tree nodes.
- **Decoupling the complicated dependencies among SIT nodes.** We analyze the dependencies of child and parent nodes in SIT, and decouple them into BMT-like dependencies via a counter-summing approach. Thus, SIT can be recovered from the consistent leaf nodes, which allows SIT to continue protecting data after system crashes and reboots.
- **Extensive experiments.** We have implemented SCUE and evaluated the designs via micro- and macro-benchmarks. The experimental results demonstrate that our proposed scheme delivers high performance while ensuring the system integrity.

II. Background

A. Threat Models

Existing designs [6], [8], [33], [44], [56], [57], [60] assume that only the on-chip domain in the computer system is safe, including the processor, cache and memory controller, which we follow in our threat model. The NVM can be attacked to reveal the data, such as stolen DIMM and bus snooping attacks [60], [61]. The encryption scheme [12] can defend the data confidentiality attacks. The memory tampering attacks modify the data in NVM to compromise the data integrity, including data replay attacks [48], i.e., attackers can use the old data to replace the new data. These integrity attacks can be detected by the integrity trees [8], [18], [19], [60], which we focus on. Other attacks are beyond the scope of this paper.

B. Counter Mode Encryption

The counter mode encryption (CME) has been widely used in state-of-the-art security systems [6], [49], [58], [62]. In general, to encrypt the data, direct AES encryption is used, which however places the decryption on the read critical path.

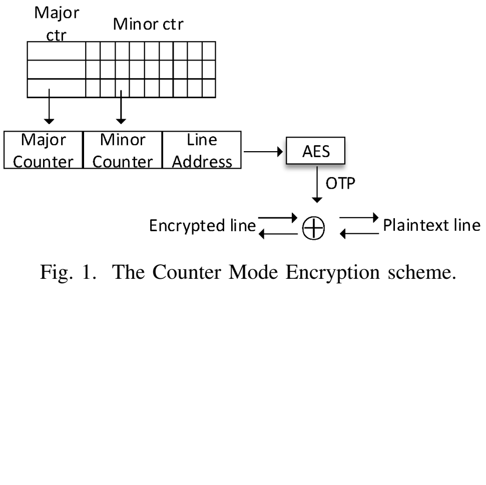


Fig. 1. The Counter Mode Encryption scheme.

Moreover, due to the unchanged secret key, the AES can be broken via the dictionary attacks [10]. To deliver high performance and improve the security of the systems, the counter mode encryption is used as shown in Fig. 1. CME uses the counter and line address to generate the one-time-padding (OTP). When writing data, the ciphertext is generated by XORing the plaintext and OTP. When reading data, since the counter blocks are cached in the memory controller, systems generate the OTP and read the encrypted data in parallel. The encrypted data is decrypted by XORing the ciphertext and OTP. Therefore, the decryption latency is masked by the read latency.

For security considerations, the OTPs cannot be reused. CME uses the line address as one input of OTP generation to ensure that different data lines have different OTPs. When one modified data is persisted into NVM, the corresponding minor counter increases by 1. For the same data, the OTP is not reused in each write since the counters are different. One counter block contains one 64-bit major counter and 64 7-bit minor counters. When the minor counter overflows, the major counter increases by 1, all minor counters in the counter block are reset to 0, and 64 corresponding user data blocks need to be re-encrypted.

C. Integrity Verification

Data integrity verification is important for system security. An attacker may tamper with the user data without authorization, i.e., integrity attacks. In general, systems use keyed-Hash Message Authentication Codes (HMACs) to verify the integrity of data [9], [36], [51]. HMACs are generated by hashing the data, address and secret key. When reading data, systems verify the data integrity via comparing the stored and recomputed HMACs. If the two HMACs are different, systems detect the unauthorized modifications. Due to the lack of secret keys, attackers fail to construct the matched HMACs of the modified data and cannot pass the integrity checking.

However, attackers may record the old data and HMAC, and use the old tuple to replace the new data and HMAC, i.e., data replay attacks. By only using HMAC, systems cannot detect the replay attacks, since the old HMAC matches the old data. To defend against the replay attacks, an integrity tree is used, including the Merkle Tree (MT), Bonsai Merkle Tree (BMT) and SGX-style Integrity Tree (SIT) [14], [20], [44], [50], to protect data.

D. Integrity Tree

1) Merkle Tree: Merkle trees are used to protect the data integrity by constructing the entire tree from the user data that are the leaf nodes in the MT. In an 8-ary MT, 8 data are hashed to generate the upper-level node that consists of 8 HMACs, called *parent node* of the hashed data. The upper-level nodes are also hashed to generate the higher-level nodes, and finally, the root is generated by iteratively hashing the leaf nodes. As shown in Fig. 2, the leaf nodes are in Level-0, i.e., leaf level, and they are iteratively hashed to construct the Level-1 and higher-level nodes. Systems store the root on the chip so that the root cannot be tampered by attackers in our threat model.

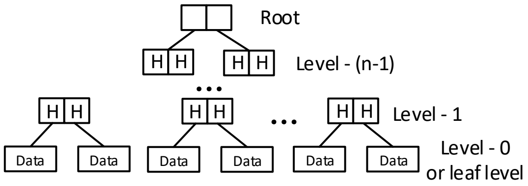


Fig. 2. The merkle tree constructed from user data.

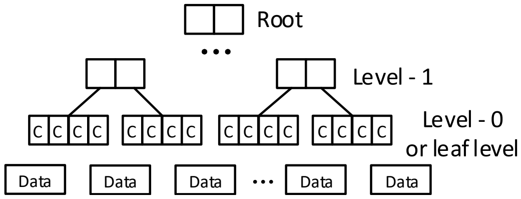


Fig. 3. The bonsai merkle tree constructed from counter blocks.

Modifying one user data results in the modification of its parent node in the MT, and the modification will propagate to the root by iteratively modifying the intermediate nodes, which are the nodes between leaf nodes and root. If attackers aim to replay one user data, they need to replay the Level-1 node. Otherwise, the replay attacks are detected by the Level-1 node. However, modifying the Level-1 nodes is detected by Level-2 parent nodes, and so on. Finally, achieving replay attacks needs to modify the on-chip root that is invulnerable to attackers. Therefore, the replay attacks cannot succeed in the MT-based memory systems.

2) Bonsai Merkle Tree: In NVM, due to the large capacity for storing user data, the number of leaf nodes in MT is large, causing a high MT. The overheads of storing tree nodes and propagating modifications from leaf nodes to root are expensive. To reduce the overheads and ensure data confidentiality and integrity, MT is combined with the CME in the secure NVM systems. The counter blocks in CME are leaf nodes in the integrity tree, and iteratively hashed to generate the root as shown in Fig. 3. This integrity tree is called Bonsai Merkle Tree (BMT) [44]. Since one counter block covers 64 data blocks, the number of the leaf nodes and the height of BMT are much smaller than those in MT. In BMT, since the data are connected with counters via the HMACs, if attackers replay data, they need to replay the counters. Therefore, the replay attacks are detected by BMT like that in MT.

3) SGX-style Integrity Tree: SGX-style Integrity Trees (SIT) [14], [45], [50] also combines MT with CME. Unlike the nodes consisting of HMACs in the BMT, as shown in Fig. 4, one tree node in SIT often consists of eight 56-bit counters and one 64-bit HMAC. The count range of a 56-bit counter, i.e., $2^{56} \approx 10^{16}$, is much larger than the endurance of NVM, i.e., 10^7–10^{12} [29], [30], [40], [59]. Therefore, in general, the counter does not overflow during the lifetime of an NVM. In SIT, one tree node has eight child nodes corresponding to eight counters in the node one by one. Moreover, the SIT node can contain more counters [45]. Without loss of generality, we follow the configurations in existing schemes [5], [60] to store 8 counters in one node in SIT. The counter blocks in CME are leaf nodes in SIT. The counter blocks (i.e.,

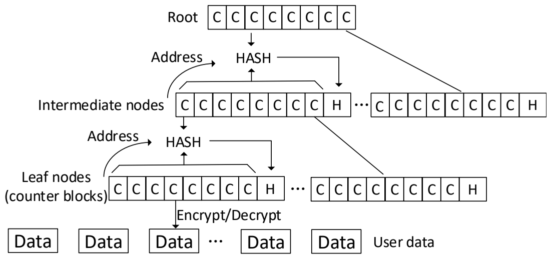


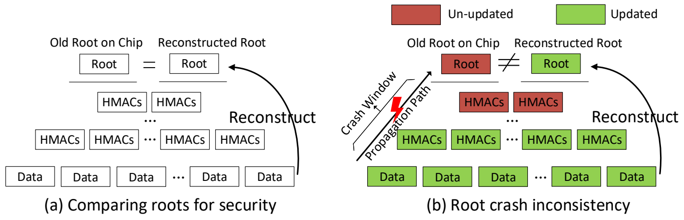
Fig. 4. The SGX-style integrity tree (SIT).

leaf nodes) and other integrity tree nodes are called security metadata, which are stored in NVM and partly cached into the metadata cache in the memory controller during the running time for improving system performance. As shown in Fig. 4, we compute the HMAC in the node by hashing the address of the node, all counters in this node, and one corresponding counter in the parent node. Persisting one modified node in SIT causes the corresponding counter in its parent node to increase by one, and the HMAC is also recomputed. Unlike other SIT nodes, the root is always in the on-chip domain, and does not contain HMAC.

4) Eager and lazy update schemes of integrity tree: For updating the root, BMT and SIT demonstrate two update schemes: eager and lazy schemes [5]. The eager scheme forces to propagate the modifications from the leaf nodes to the root once the modified leaf nodes are persisted. In the eager scheme, persisting one modified leaf node incurs the changes on all nodes in the branch, including the root. For BMT, all HMACs in the propagation path are re-hashed, and for SIT the corresponding counters and HMACs in these nodes are updated. In the lazy scheme, when one node of SIT/BMT is flushed into NVM, the system only updates its parent node by modifying the corresponding counter/HMAC. Other ancestor nodes, including the root, are only updated when their child nodes are flushed into NVM. Therefore, the root is not immediately updated in the lazy scheme.

Both eager and lazy schemes ensure the integrity of user data during running time. Since the cache is in the secure domain, the cached nodes are treated as the trusted bases like the root to verify other data. To implement the transmission of the trusted base from root to the cached nodes, each data entering cache must be verified. When reading data, systems read the ancestor nodes until one ancestor node has already existed in cache. These ancestor nodes are iteratively verified to ensure the security of the read data.

SIT uses counters and HMACs to protect data. When the user data and HMAC are read, systems recompute the HMAC by hashing the address, the data and the parent counter in counter block to verify the integrity of data. Attackers also replay the counter block when persisted in NVM. However, the parent node of the counter block is updated in cache. The replayed counter block needs to be verified by the updated parent node when the counter block re-enters the cache. In the worst case, attackers tamper with all nodes in a branch to execute the replay attacks. However, since the root always resides in the chip, the updated root is used to iteratively verify the integrity of the user data.



(a) Comparing roots for security

(b) Root crash inconsistency

Fig. 5. The integrity tree w/o and w/ root crash inconsistency. (a) The system compares the reconstructed and old roots to verify the recovery process. (b) When crashes occur during Crash Window, the upper nodes and root are not updated. The un-updated root cannot verify the recovery process.

In SIT, once counters have been increased, the HMACs of different nodes in the branch can be computed in parallel while BMT sequentially computes the HMACs. Moreover, as shown in Fig. 4, one counter in SIT is protected by 2 HMACs, i.e., the HMACs in the node itself and child node. SIT is more secure than BMT [5]. Therefore, in this paper we focus on SIT instead of BMT to protect data integrity.

III. The Problem of Updating Root in SIT

A. The Trust Base of Integrity Verification

In the integrity tree, only the root is always stored in the secure on-chip domain. Other tree nodes may be flushed into NVM and illegally modified by attackers. Therefore, the root is the trust base of integrity verification.

In DRAM, since the data are lost after system crashes, existing integrity trees are discarded. A new integrity tree is further constructed from the new data in DRAM. Unlike DRAM, due to the non-volatility of NVM, after system crashes and recovery, the data are still maintained in the NVM, and the integrity trees are also used to verify the data. Therefore, the integrity trees need to be recovered from system crashes. The root of an integrity tree in the on-chip non-volatile register plays the role of trust base during system recovery.

Existing schemes have widely discussed how to recover the integrity trees [5], [8], [23], [60]. The basic idea of recovering the integrity tree is to reconstruct the tree from leaf nodes up and compare the reconstructed root with existing tree root. As shown in Fig. 5(a), we take an MT as an example. The system iteratively hashes the user data in NVM to reconstruct the integrity tree. The old root in the on-chip non-volatile register is the root of the integrity tree before system crashes. The reconstructed root is further compared with the old root to verify the correctness of the recovery process. If attackers illegally modify the data, the reconstructed root does not match the stored one, and the system reports the attacks.

Existing schemes [5], [23], [60] propose different approaches to speeding up the recovery process of the integrity tree and do not need to reconstruct the tree from leaf nodes. However, they also rely on a side-tree to verify the correctness of recovery, which needs to reconstruct the tree and compare the reconstructed root with the old one, e.g., the shadow tree in Anubis and Phoenix [5], [60], and cache tree in STAR [23].

B. The Problem of Root Crash Inconsistency

After system crashes and recovery, the root of the integrity tree is important for system security. However, the root cannot be used as the trust base after system crashes due to the **root crash inconsistency**, which is described below.

Persisting user data from cache into NVM results in the modification of their corresponding leaf nodes of the integrity tree in cache. Propagating the modifications from the leaf nodes to the root needs a long time to update all intermediate nodes on the propagation path. The propagation time is called the *root crash inconsistency window*, or *crash window*. As shown in Fig. 5(b), if a system failure occurs during the *crash window*, the propagation process of the modifications aborts. The new user data are persisted into NVM, but the root of the integrity tree is not updated. The root is inconsistent with the user data upon crashes, i.e., *root crash inconsistency*. Once system recovery, the tree is reconstructed from the leaf nodes, and a new root is constructed via the updated user data. The old un-updated root stored on-chip is not equal to the new root, thus leading to recovery failure.

Both lazy and eager update schemes suffer from the *root crash inconsistency* problem. For a lazy update scheme, the root is *lazily* updated and always inconsistent with the user data upon crashes. For an eager update scheme, it is potential to achieve root crash consistency due to immediately updating the root. However, although the latency of propagating modifications from leaf nodes to root is low due to computing the HMACs in parallel in SIT (about 40 cycles [18], [19]), a system failure occurring during the 40-cycle *crash window* incurs the root crash inconsistency problem as shown in Fig. 5(b).

Due to the root crash inconsistency, the root cannot be used as trust base during system recovery. As shown in Fig. 5(b), the system reconstructs the integrity tree based on the updated data in NVM. However, the reconstructed root does not match the un-updated root in the on-chip non-volatile register. Therefore, the system reports attacks, and the recovery always fails even though no attacks occur.

Existing designs, say PLP [18] and BMF [19], discuss how to ensure the root crash consistency in BMT. But PLP and BMF do not focus on SIT due to the complicated dependencies (described in §III-D) among SIT nodes [18], [19].

C. Discussions about ADR/eADR

Intel has proposed eADR [2] in their new processor. The eADR, together with ADR, has the ability to flush the data from caches (including CPU caches and metadata cache) and write pending queue (WPQ) into NVM upon system crashes. However, when crashes occur during *crash window* as shown in Fig. 5(b), the leaf nodes and some low-level intermediate nodes are updated, but the high-level nodes and root are not updated. The leaf nodes and intermediate nodes (no matter whether they are updated or not) in cache are flushed into NVM by the support of eADR, but the root is still not updated as described in §III-B. For updating the root upon system crashes, the intermediate nodes need to be read into the cache from NVM to propagate the modifications after system crashes, which are not supported by eADR. Even if we have read all intermediate nodes on the propagation path into a large cache before crashes, updating the root needs to compute the HMACs in the intermediate nodes after crashes, which is not supported by eADR. Hence, to address the problem of root crash inconsistency, instantaneously updating the root is still necessary even in the eADR systems.

Reading data and computing HMACs upon system crashes via the energy of eADR seem promising. However, eADR is not only the backup energy but also a mechanism on the OS [2], [3] to detect crashes, flush data to correct location in NVM, etc. Updating the root upon crashes is not supported by current eADR and requires significant modifications in the OS-level mechanism of eADR.

D. The Problem of Reconstructing Root in SIT

Due to high security and performance [5], in this paper we focus on SIT to protect the security of data in NVM. However, we observe that SIT root cannot be reconstructed from leaf nodes due to the complicated dependencies among SIT nodes [18], [19], [60], and SIT cannot continue to protect data after system crashes and recovery.

In BMT, one node is computed by its child nodes, i.e., high-level nodes depend on low-level nodes. Thus, BMT can be reconstructed from leaf nodes up. Unlike BMT, as shown in Fig. 4, constructing one tree node in SIT needs its parent node as input, i.e., low-level nodes depend on high-level nodes. Since some intermediate nodes have not been updated even if using eADR upon system crashes, the whole tree of SIT and the root cannot be reconstructed from the leaf nodes after the system reboots [8], [18], [19], [60]. STAR [23] and Anubis [60] propose to recover SIT, but they do not consider root crash consistency. PLP [18] and BMF [19], which ensure root crash consistency of BMT, choose to update the root, intermediate nodes, and leaf nodes in an atomic approach to ensure the crash consistency in SIT with high overheads.

Without the reconstructed root, the stored root on-chip cannot be used to detect the attacks in leaf nodes. **Therefore, even if the root in SIT is correctly updated in time, the system cannot use the root to verify the data integrity after the system reboots.** In this paper, we decouple the complicated dependencies among SIT nodes. Thus, the SIT can be reconstructed from leaf nodes up like BMT. The consistent leaf nodes and root are enough to ensure data integrity after system failures and reboots.

In summary, when using SIT, there are two problems, i.e., *how to ensure root crash consistency, and how to reconstruct the tree (including root) from leaf nodes.* To improve system performance and correctly update the root, we propose SCUE, a low-overhead shortcut update scheme based on SIT, to ensure the root crash consistency by instantaneously updating the root. SCUE also decouples complex dependencies between SIT nodes and their parent nodes so that SIT can be reconstructed from leaf nodes.

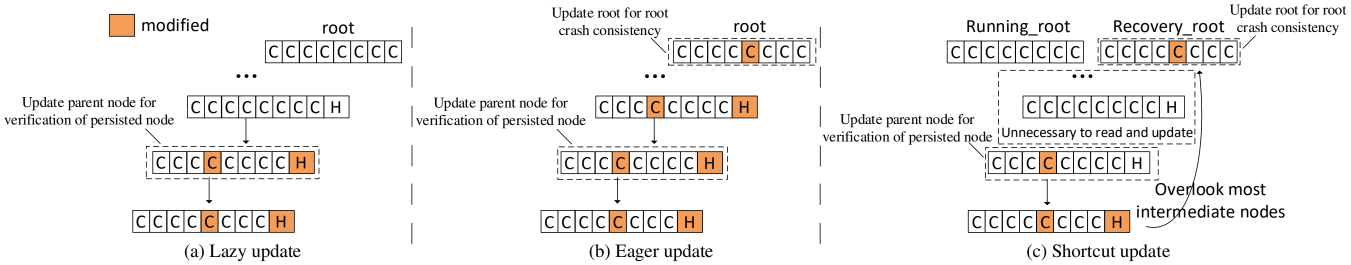

modified
root
C C C C C C C C
Update parent node for verification of persisted node
C C C C C C C H
(a) Lazy update
Update root for root crash consistency
root
(b) Eager update
Running_root
Recovery_root
Update root for root crash consistency
Unnecessary to read and update
Update parent node for verification of persisted node
Overlook most intermediate nodes
(c) Shortcut update

Fig. 6. The Lazy, Eager and SCUE schemes in SIT. (a) Lazy update scheme modifies the parent node for integrity verification. (b) Eager update scheme modifies the intermediate nodes and root. (c) SCUE leverages two roots, which actively overlooks most intermediate nodes and only modifies parent node and Recovery_root.

IV. The Design of SCUE

The crash consistency among user data, counter blocks, and HMACs in user data has been widely researched by legacy designs [18], [19], [50], [57], [61]. Unlike them, we focus on the root crash consistency in SIT by proposing SCUE to instantaneously update the root (§IV-A) and decouple the complex dependencies among SIT nodes (§IV-B). SCUE is orthogonal to these crash consistency schemes [50], [57], [61] (except PLP and BMF [18], [19] that only focus on BMT).

A. Update the Root in a Shortcut Manner

Although all HMACs in the updating branch of SIT can be updated in parallel in about 40 cycles [18], [19], the 40-cycle *crash window* still exists in the modification propagation path from leaf node to root, leading to crash inconsistency problem. Moreover, propagating the modifications needs to read the intermediate nodes in the propagation path from NVM into cache (if they are not in cache due to the limited size of cache), which extends the *crash window*. We call the latency of calculating HMACs in the propagation path the *hash_latency*, and the latency of reading intermediate nodes from NVM to cache the *read_latency*, both of which together constitute the *crash window*.

To eliminate the *crash window*, we propose *a low-overhead **S**hort**C**ut **U**pdat**E** (SCUE) scheme*. The idea of SCUE is to efficiently remove the computations of HMACs and the reads of the intermediate tree nodes from the write critical path in SIT. Moreover, since the root is always *lazily* updated, root crash inconsistency is inevitable in a lazy update scheme. Our SCUE focuses on the eager update scheme to instantaneously update the root.

1) Lazy computing in SCUE: We observe that in one SIT node, the HMACs are used to verify the node itself, and the counters are used to verify the child nodes. When reading data from NVM into cache, the counters in the cached ancestor tree nodes and HMACs in the read data are used to verify the integrity of the read data. Since the on-chip data are invulnerable to attackers in our threat model, the integrity of the cached tree nodes does not need to be verified, and the HMACs in the cached data are not used. To verify the integrity of the data the next time they are read, we increase the counters in the cached ancestor tree nodes when the user data are persisted, but updating HMACs is unnecessary. Only the HMACs in the persisted data need to be updated to defend against attacks that occur in NVM.

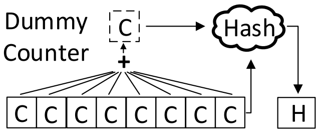


Fig. 7. Dummy counter is generated by summing all counters in the tree node.

We propose Lazy Computing in SCUE to remove the *hash_latency* in the propagation path from leaf nodes to root. When one leaf node is persisted, the HMAC in the leaf node is computed to facilitate the verification of the following read. The counters in the ancestor tree nodes (including the root) are updated by increasing 1, but the HMACs of the ancestor tree nodes are only computed when the nodes are flushed into NVM via the cache replacement policy.

2) Remove the reads of tree nodes: Propagating the modifications from leaf nodes to root needs to read the ancestor nodes of the leaf nodes with long *read_latency*. We now describe how to reduce the *read_latency*. As shown in Fig. 6(a), the lazy update scheme only updates the parent node of the persisted node for verification as described in §II-D4. The root is not updated in the lazy update scheme, thus suffering from *root crash inconsistency* problem. In SIT, the root consists of eight counters, and the counter values are related with the leaf nodes. In the eager update scheme as shown in Fig. 6(b), when one leaf node is modified, its ancestor nodes are updated. Specifically, all corresponding counters in nodes increase by 1, including the counter in root. Therefore, if one modified leaf node is persisted into NVM, the corresponding counter in root increases by 1. To eliminate the *crash window* described in §III-B, as shown in Fig. 6(c), we propose the SCUE with a *Running_root* and a *Recovery_root* (stored in the non-volatile register on-chip) for system running and recovery respectively. SCUE updates the parent nodes for verifying the persisted node without HMAC computation, and directly increases the counter in Recovery_root for root crash consistency without reading the intermediate tree nodes in the propagation path.

When the cached tree nodes are flushed into NVM due to cache replacement policy, their parent nodes need to be read to compute the HMACs of the persisted nodes, which possibly incurs iterative reads of ancestor nodes to execute the integrity verification. To reduce the number of these reads, SCUE proposes the *dummy counter* as shown in Fig. 7. In the eager scheme, when any counter in one node increases by 1, the corresponding counter in the parent node (called *parent counter*) also increases by 1, i.e., the parent counter is the sum of all counters in the child node. When flushing one tree node from cache into NVM, the system generates a dummy counter by summing all counters in the flushed tree node with low overheads. Therefore, the dummy counter becomes the parent counter of the flushed node to compute the HAMC. By using *dummy counter*, in our SCUE, the parent node of the persisted node does not need to be read when flushing tree nodes. The iterative reads of the ancestor nodes on write critical path are also removed.

When writing a leaf node, our SCUE generates the dummy counter to compute the HMAC of the persisted leaf node, and forces to directly update the Recovery_root (i.e., increasing the corresponding counter by 1) by overlooking the intermediate nodes. After updating the Recovery_root, the write operation is completed. We then update the parent node of the persisted data using the dummy counter. The updates of intermediate nodes in SCUE are similar to the lazy scheme described in §II-D4, i.e., when one modified node is persisted from cache into NVM, the parent counter is updated to the sum of all 8 counters of the persisted node. Specifically, assuming a leaf node A is to be persisted into NVM, SCUE generates the *dummy parent counter* to compute the HMAC of node A (if its parent node B is not in cache), and the corresponding counter in Recovery_root increases by 1 at the same time. The *crash window* is thus removed. Moreover, after the flush of A has been completed, we force to read B with iterative reads of the ancestor nodes (the B is verified by the ancestor nodes). The parent node B is thus updated via the generated *dummy counter*. These operations are off the critical path of data write and do not extend the *crash window*. In SCUE, the node is updated when its child node is flushed into NVM like the lazy update scheme. The only difference is that the lazy update scheme updates the parent counter of the flushed node by increasing the value of the old parent counter by 1, but SCUE updates the parent counter via the sum of 8 child counters. In most cases, after increasing the old parent counter, the value of the parent counter is equal to the sum of 8 child counters, unless the child counters have been modified more than once before the child node is flushed from cache into NVM. The Running_root is also updated by summing 8 counters of its modified child node (i.e., the dummy counter) when its child nodes are persisted into NVM, like the lazy update scheme.

In summary, in SCUE, the root is instantaneously updated by increasing the corresponding counter in root by 1 when the leaf node is persisted. Thus, the *crash window* is completely removed.

3) Security analysis: SCUE updates the Running_root and integrity tree like the lazy update scheme, i.e., updating the parent nodes when persisting data. Therefore, our SCUE provides the same security guarantee as the lazy update scheme. In our SCUE, if attacks occur in the data in NVM, the attacked data are verified by the updated parent node in cache. When reading data, SCUE uses Running_root to iteratively verify the read data if all ancestor nodes are not in cache. The Running_root is used for data security during system running, and the Recovery_root is used for the integrity tree recovery as shown in §IV-B.

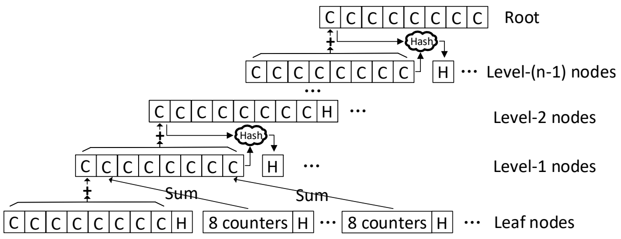


Fig. 8. Reconstructing the SIT from leaf nodes.

B. Reconstruct the SIT

As described in §III-D, low-level nodes depend on high-level nodes in SIT. Even if the consistency among root and leaf nodes is guaranteed, SIT fails to be reconstructed from the leaf nodes [5], [18], [60], since reconstructing one SIT node requires its parent node as input. The root thus cannot verify the integrity of leaf nodes after the system crashes and reboots. In this section we decouple the dependencies among SIT nodes.

1) Counter-summing for reconstructing SIT: In SCUE, although the Recovery_root is instantaneously updated, due to failing to reconstruct the integrity tree from the leaf nodes, the up-to-date Recovery_root in SIT fails to verify the leaf nodes after system reboots. To provide efficient integrity verification for SCUE, we re-think the relationships between parent nodes and child nodes in SIT, and use the counter-summing recovery to reconstruct the SIT.

In SIT, each tree node contains eight counters and has eight child nodes. We observe that in eager update scheme, when one child node is modified, i.e., any counter in the node increases by 1, its parent counter also increases by 1. Therefore, the parent counter becomes equal to the sum of all 8 counters in the child nodes. The root counter further becomes equal to the sum of all counters in the leaf nodes covered by the root counter. Now we can use the sum of counters in the child node to update the parent counter in our SCUE as described in §IV-A2. As shown in Fig. 7, the dummy counter is generated according to this observation. After the system reboots, we reconstruct the tree from the bottom up via the dummy counter.

We first reconstruct the Level-1 nodes (which are parent nodes of leaf nodes) from the consistent leaf nodes. Reconstructing one SIT node requires correct *counters* and *HMAC* in the node.

As shown in Fig. 8, to reconstruct the *counters*, we generate the dummy counters from all leaf nodes. We aggregate the $8K$th to $8K+7$th (K is a natural number) dummy counters to form the Kth Level-1 node, thus constructing counters in the reconstructed nodes. We then use the HMACs in the persistent leaf nodes to verify the correctness of the reconstructed counters in Level-1 nodes. The HMAC is recomputed by hashing the reconstructed counter in the Level-1 node and all counters in the corresponding leaf node. The mismatches of recomputed and stored HMACs in the leaf nodes indicate the attacks occurring during system recovery. All counters in the Level-1 nodes can be reconstructed via summing the corresponding counters in the child leaf nodes.

For recomputing *HMACs* in Level-1 nodes, the counters in Level-2 nodes (which are parent nodes of Level-1 nodes) need to be reconstructed by generating the dummy counters from Level-1 nodes. The HMACs in the Level-1 nodes are recomputed by hashing the corresponding counters in the Level-2 nodes and all counters in the Level-1 nodes as shown in Fig. 8. Therefore, the counters and HMACs in the Level-1 nodes are reconstructed. We use the same way to reconstruct the whole tree from the bottom up. Finally, the dummy Recovery_root counters are generated. If the dummy Recovery_root counters are different from the stored Recovery_root counters, the SIT reconstruction fails, and attacks occur during recovery. Otherwise, SIT is successfully reconstructed. The reconstruction process is similar to that of MT/BMT, i.e., one tree node is recovered from its all child nodes.

In summary, SCUE decouples the dependencies among nodes in SIT by exploring and exploiting the observation that the parent counter is the sum of the child counters, which allows to efficiently reconstruct the parent nodes from the child nodes.

2) Security analysis: We analyze the security of the proposed counter-summing recovery approach in SIT. We focus on the Recovery_root and leaf nodes since the intermediate tree nodes are lost or stale when failures occur, which need to be reconstructed from the persistent leaf nodes. Moreover, since the Recovery_root is on chip, attackers cannot tamper with the Recovery_root, and only the leaf nodes in NVM are attacked.

We use the HMACs in the leaf nodes and the on-chip Recovery_root to detect the attacks that occur during system recovery. All attacks in the leaf nodes can be divided into two types: *roll-forward* and *roll-back* attacks. Specifically, the roll-forward attacks tamper with the counter value in the leaf nodes to a larger value. On the contrary, the roll-back attacks tamper with the counter value in the leaf nodes to a smaller value. The replay attack is a special roll-back attack that replaces the counters and HMACs with the old tuple.

As shown in Table I, the roll-forward attacks are detected by the HMACs in the leaf nodes. The parent counters are reconstructed from the leaf nodes, and the HMACs of the leaf nodes are recomputed. Without the secret key, attackers cannot calculate the correct HMACs using new counters. The mismatches between the recomputed and stored HMACs in leaf nodes indicate the existence of attacks.

HMACs detect roll-back attacks (except the replay attacks) due to not matching the tampered counters. We discuss how to detect the replay attacks, which pass the verification of HMAC since the old HMAC matches the old counters. Since increasing counters in the leaf nodes causes the increment of counters in Recovery_root, one Recovery_root counter becomes equal to the sum of all counters in the corresponding leaf nodes. For example, the value of the first counter in the Recovery_root is the sum of all counters' values in the first 1/8 of the nodes in the leaf level since they are covered by the first root counter. Rolling back counters in leaf nodes causes the counter in the reconstructed Recovery_root to mismatch the counter in the stored Recovery_root. Therefore, the attacks are detected. Moreover, attackers may roll back and roll forward some counters in the leaf nodes at the same time to mislead the Recovery_root, but rolling forward counters is detected by the HMACs as shown in Table I.

TABLE I
THE ATTACKS CAN BE DETECTED BY HMACS AND ROOT.

	Roll-forward attacks	Roll-back attacks	Roll-forward + roll-back attacks
HMACs in leaf nodes	detected	detected	detected
Recovery_root	/	detected	/

TABLE II
THE CONFIGURATIONS OF THE EVALUATED NVM SYSTEM.

Processor	
CPU	8 cores, X86-64 processor, 2 GHz
Private L1 cache	64KB, 2-way, LRU, 64B Block
Private L2 cache	512KB, 8-way, LRU, 64B Block
Shared L3 cache	4MB, 8-way, LRU, 64B Block
DDR-based PCM Main Memory	
Capacity	16GB
PCM latency model	tRCD/tCL/tCWD/tFAW/tWTR/tWR =48/15/13/50/7.5/300 ns
Write queue	64 entries with tags for user data, 10 entries without tags for security metadata
Secure Parameters	
Security metadata cache	256KB, 8-way, 64B Block, in MC
SIT	9 levels, 8-ary, 64B Block
Hash latency	{20, 40, 80, 160} cycles (default 40)

The detection process of SCUE is similar to that of MT/BMT, i.e., the correctness of reconstructing the tree is verified after reconstructing the whole integrity tree and comparing the reconstructed Recovery_root with the stored one.

V. PERFORMANCE EVALUATION

A. Evaluation Methodology

To evaluate the performance of SCUE, we model the system in Gem5 [11] and NVMain [39] which is a cycle-accurate main memory simulator for NVM technologies. NVMain is co-compiled with the Gem5 and takes over the memory part of Gem5. This Gem5-NVMain co-simulator has been widely used in existing works [23], [24], [32], [52], [54], [61], [62]. The main parameters are shown in Table II. The metadata cache in the memory controller is 256KB for storing counter blocks and integrity tree nodes. We model the PCM technologies by using 16GB capacity. The PCM latency is modeled like existing designs [23], [55], [61]. In general, the hash latency to generate the HMAC is 40 cycles [18], [19]. In our sensitive study, we vary the hash latencies (from 20 to 160 cycles [20], [34], [47]) to analyze the impacts on performance. Our SCUE is compared with PLP [18] and BMF [19]. To facilitate fair comparisons, we use 8 representative applications from CPU

SPEC2006 benchmarks [22], which have been used in PLP and BMF. These applications come from integer and floating-point fields with different programming languages (C, C++, and Fortran) and contain about 50% memory instruction (load and store). They are fast forwarded to representative regions, and we further run 5 billion instructions for each application after 10 million instructions warm-up in the syscall emulation (SE) mode of Gem5. Moreover, for comprehensive evaluation, we also use 5 persistent workloads to evaluate the systems. The persistent workloads, i.e., array, btree, hash, queue and rbtree, are widely used in existing schemes of persistent memory [13], [23], [26], [27], [33], [43], [61].

To comprehensively examine the performance of our proposed SCUE, we evaluate and compare the following schemes.

- **Persist Level Parallelism (PLP).** PLP [18] focuses on BMT to provide root crash consistency. PLP is an eager update scheme to propagate the modifications from leaves to root in pipeline. Due to the complicated inter-level dependence in SIT, PLP needs to update and persist the ancestor nodes of the persisted leaf nodes when leveraged in SIT, incurring high overheads.
- **Ideal case of BMF (BMF-ideal).** BMF [19] chooses some intermediate nodes to be stored in the non-volatile metadata cache (nvMC) as persistent roots. BMF updates the tree nodes until the node is in nvMC. In the ideal case of BMF, the size of nvMC is unlimited and all counter blocks' parent nodes exist in the nvMC, i.e., all of them are persistent roots. When used in SIT, BMF needs to persist all intermediate nodes from leaf to root for verification after system reboots.
- **Lazy scheme (Lazy).** The lazy scheme updates the parent nodes of the written data in the integrity tree but does not propagate the modifications to the root. It also needs to read the ancestor nodes to verify the parent node when writing data. **Lazy scheme does not guarantee root crash consistency.**
- **Our shortcut update scheme (SCUE).** SCUE directly updates the root without propagating the modifications from leaf nodes to root and ensures the root crash consistency. In SCUE, the dependencies among SIT nodes are decoupled into the BMT-like dependencies.
- **Insecure baseline system (Baseline).** Insecure baseline system only contains the counter mode encryption to encrypt the user data without integrity verification.

B. Results and Analysis

Propagating the modifications from leaf nodes to the root on the write critical path needs to read and update the intermediate nodes in the branch, incurring a long latency. We evaluate the write latencies and execution time on different schemes.

Fig. 9 shows the write latencies on different schemes. PLP is inefficient in SIT. In addition to computing multiple hash calculations in parallel (40 cycles), PLP needs to read, update and persist the shadow copies of intermediate nodes when writing data in the SIT-based system. On average, the write latency in PLP is 2.74x than that in Baseline. The Lazy scheme needs to read the parent and ancestor nodes of the evicted data, and calculate the HMACs in the evicted data and parent nodes on the write critical path. Since all counter blocks' parent nodes are persistent roots, BMF-ideal does not need to flush intermediate nodes like PLP, and only updates the counter blocks and parent nodes when writing data. Our SCUE generates a dummy counter and only computes one HMAC, without reading any nodes when writing data. Therefore, the write latency in SCUE becomes lower than that in the Lazy/BMF-ideal scheme. On average, the write latencies in the Lazy/BMF-ideal/SCUE scheme are 1.29x/1.21x/1.12x than that in Baseline.

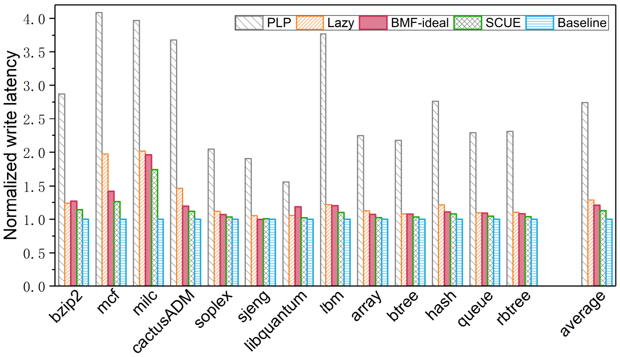


Fig. 9. The write latencies on different workloads (normalized to Baseline).

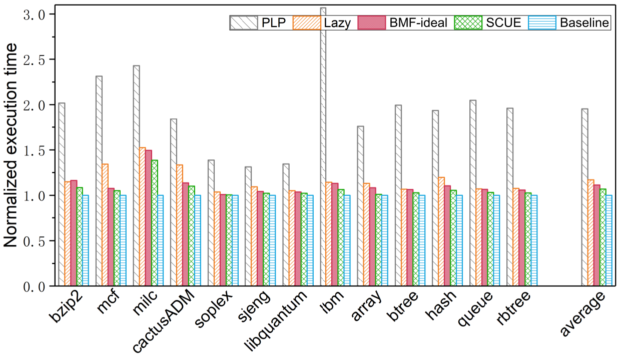


Fig. 10. The execution time on different workloads (normalized to Baseline).

As shown in Fig. 10, SCUE shows lower execution time than the PLP, Lazy and BMF-ideal schemes. Specifically, the PLP leads to a 1.96x slowdown than Baseline on average due to persisting the shadow copies of the updated intermediate nodes, while SCUE produces to a 1.07x slowdown. SCUE also delivers higher performance than the Lazy scheme in terms of execution time. The average execution time in the Lazy scheme is 1.17x than that in Baseline, while SCUE is able to verify the integrity after the system reboots and the Lazy scheme fails. Like the write latency in Fig. 9, for different workloads, BMF-ideal demonstrates different performance. But on average, BMF-ideal shows a similar execution time to our SCUE, i.e., 1.11x slowdown than Baseline.

The improvements in terms of write latency and execution time in SCUE compared with other schemes come from removing the reads and writes of intermediate nodes on the write critical path. In SCUE, writing a leaf node requires computing the HMAC in the leaf node and instantaneously increasing the corresponding counter in the root. Unlike SCUE, PLP needs

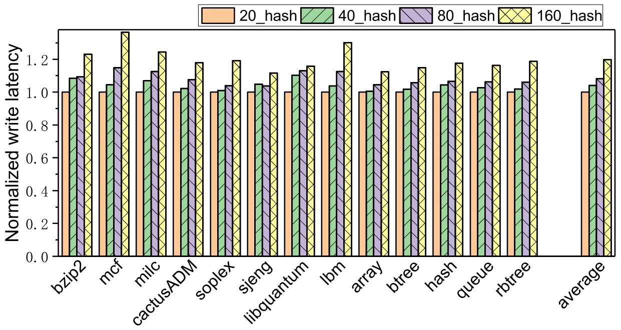


Fig. 11. The write latencies of using different hash computations in SCUE. The 20_hash, 40_hash, 80_hash and 160_hash respectively represent the needed 20/40/80/160 cycles (normalized to the 20_hash).

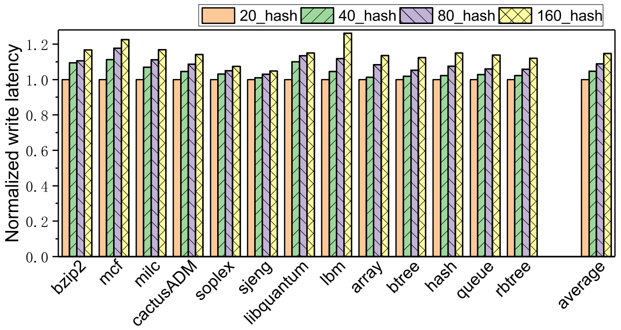


Fig. 12. The execution time of using different hash computations in SCUE. The 20_hash, 40_hash, 80_hash and 160_hash respectively represent the needed 20/40/80/160 cycles (normalized to the 20_hash).

to read the ancestor nodes to propagate the modifications to the root. Lazy and BMF-ideal need to read the parent node to compute HMAC in the persisted leaf node and ancestor nodes for verifying the parent node. Due to the complicated dependencies in SIT, the updated intermediate nodes need to be persisted for crash consistency [18], [19].

C. The Sensitivity to the Hash Latency

Existing configurations allow the hash latency to generate the HMACs to be 40 cycles. To further examine the performance in the larger scale, we set the hash latency from 20 cycles to 160 cycles like existing works [20], [34], [47].

As shown in Fig. 11, increasing the hash latency (from 20 to 160 cycles) incurs on average 1.20x (up to 1.36x) write latency increase in SCUE. Fig. 12 shows the execution time when adjusting the hash latency. Since we reduce the number of hash calculations to one when writing data, the execution time only increases by 1.14x when the hash latency is 160 cycles. This result shows that even if we use more secure hash algorithms with higher computation latency (e.g., 160 cycles) [21], SCUE not only protects systems, but also delivers comparable performance.

D. Time overhead of recovering SIT in SCUE

In our SCUE, the parent node in SIT can be constructed from child nodes like MT/BMT. Our SCUE for SIT is hence orthogonal to the BMT/SIT recovery designs [5], [8], [23], [60]. For example, we can leverage the *bitmap lines* in STAR [23] to indicate the stale nodes in the SIT of SCUE with small write overheads (SCUE-STAR), and generate the *dummy counter* from child nodes to reconstruct the stale nodes. As shown in Fig. 13, for a 4MB metadata cache, the recovery time is about 0.05s. We can also leverage the *shadow table (ST) block* in Anubis [60] to record the addresses of stale nodes in our SCUE (SCUE-AGIT). Since our SCUE is able to reconstruct the tree from leaf nodes up, although SCUE is designed for SIT, the *ST block* only records the addresses without the contents of stale nodes like AGIT (Anubis for General Integrity Tree) not ASIT (Anubis for SGX-style Integrity Tree) in Anubis (the stale node is reconstructed from child node via dummy counter like BMT) with small overheads instead of 2x write overheads [23], [60]. The recovery time is about 0.17s for a 4MB metadata cache. The evaluation of recovery time is similar to existing works [23], [57], [60], i.e., we assume that fetching one metadata from NVM consumes 100ns, and the latencies of reading metadata from NVM dominate the recovery time. Note that the recovery time of SCUE-AGIT in Fig. 13 is shorter than that reported in Anubis, since one node in SIT of SCUE contains only 8 counters instead of 64 counters in the General Integrity Tree (GIT) in Anubis.

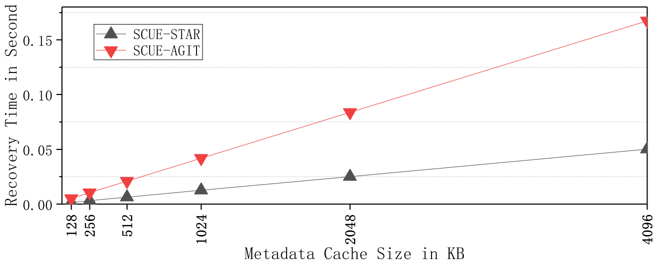


Fig. 13. The recovery time when leveraging STAR/AGIT to recover the SIT in SCUE.

However, STAR/Anubis needs a cache tree/shadow tree (called side tree) to ensure the correctness of the recovery process [23], [60]. These side trees need to be reconstructed from leaf nodes up and compare the reconstructed root with the old one to verify the recovery process [23], [60], which may fail since the root is inconsistent with the persistent leaf nodes after crashes as described in §III-B, leading to the recovery failures in STAR and Anubis. We use SCUE to ensure the root crash consistency.

In this paper, we do not consider the fast recovery since existing fast recovery designs [5], [8], [23], [60] are orthogonal to our SCUE. We aim to address the problem of root crash inconsistency.

E. The number of memory accesses

Memory accesses consist of user data and security metadata memory accesses. Due to the use of metadata cache, user data accesses dominate the memory accesses. The numbers of user data memory accesses in all schemes are the same when running the same applications. For security metadata, although our SCUE removes the metadata reads from the write critical path, SCUE demonstrates the approximate numbers of memory accesses to the Lazy scheme. The reason is that when reading user data, the ancestor nodes need to be read in

SCUE and Lazy. For PLP, since it needs to read and persist the intermediate nodes when being leveraged in SIT, PLP incurs about 7.04x memory accesses than Lazy on average for our 9-level SIT. For BMF-ideal, since all parent nodes of counter blocks are roots, BMF-ideal eliminates the memory accesses of ancestor nodes. However, BMF-ideal still needs to read and write counter blocks, which dominates the security metadata memory access [5]. BMF-ideal reduces about 8.7% memory access compared with Lazy on average.

F. Space and hardware overheads

Besides security metadata cache that is required in every secure NVM system [5], [8], [18], [19], [23], [60], our SCUE leverages *two 64B on-chip non-volatile registers (128B)* to store the Running_root and Recovery_root in §IV-A2. PLP relies on *PTT (616B) and ETT (48b)* to respectively support pipelined and out-of-order (OOO) tree updates.

The BMF-idea requires a large non-volatile metadata cache (nvMC) to store the counter's parent nodes as the persistent roots. For a 16GB NVM, the size of nvMC in BMF-idea is *256MB*. In BMF's paper [19], they also reported *512b to 16MB* nvMC in the non-ideal BMF scheme.

VI. Discussions

The on-chip overheads of BMF. Bonsai Merkle Forests (BMF) [19] divides a big BMT into multiple small BMTs. In the ideal case of BMF (BMF-ideal), these small trees are two-level, i.e., eight leaf nodes and one root without intermediate nodes. BMF-ideal does not need to propagate the modifications and persist the intermediate nodes, thus demonstrating similar performance to our SCUE. However, for security, the roots of these small trees are always stored in the on-chip cache (Section 4.4 in BMF paper [19]). The cache composed of SRAM with eADR is not enough since the roots in the cache will be persisted into NVM upon crashes, thus violating the security requirement. The nvMC in BMF needs to be composed of high-speed non-volatile registers, with high on-chip overheads.

Importance of decoupling the complicated dependencies in SIT. The complicated dependencies limit the use of SIT [18], [19], [60]. In this paper, we decouple the complicated dependencies in SIT into BMT-like dependencies. The BMT-based optimizations [8], [18], [19], [60] can be applied in SIT. Therefore, we strongly argue that SIT can completely replace BMT with high performance [60], high security [5], and low storage overheads [45], [50].

VII. Related Work

Security metadata recovery. Security metadata include counter blocks and integrity tree nodes. For improving performance, security metadata are cached in a volatile on-chip buffer in the memory controller. After system failures, some updates of security metadata are lost due to not being instantaneously persisted into NVM. To recover the counter blocks, Osiris [57] relaxes the persistence of counter blocks, and retrieves the counters from the stale state in NVM. Supermem [61] uses a write-through scheme to ensure the consistency of counter blocks. When the counter blocks are modified, they are directly flushed into NVM. To recover the BMT, Triad-NVM [8] persists low-level tree nodes with user data. After failures, the systems can reconstruct the BMT from the persistent low-level nodes. Anubis [60] records the address and contents of the modified cached metadata in the shadow table in NVM. According to the shadow table, Anubis recovers both BMT and SIT. STAR [23] persists the modifications of SIT in the MAC fields without extra memory writes.

Our SCUE is orthogonal to the security metadata recovery works. Osiris [57] and Supermem [61] can be used in SCUE to ensure the consistency between counter blocks and user data. Moreover, SCUE leverages STAR [23]/Anubis [60] to fast recover SIT as shown in §V-D.

Security metadata organization. The security metadata are organized in multiple ways to improve the performance of accessing metadata. VAULT [50] reduces the height and space overhead of SIT by storing more than 8 counters in one node. Based on VAULT, MorphCtr [45] observes that when one counter overflows, either less than a quarter of counters or all the counters are used. The MorphCtr scheme provides a scalable solution to store 128 counters in one node and further reduces the height of the tree. Synergy [46] places the HMAC inside the ECC chip in a 9-chip ECC-DIMMs and demonstrates that HMAC can be used to detect not only data tampering but also memory errors.

Unlike existing schemes, our SCUE instantaneously updates the root, removes the *crash window* by overlooking the intermediate nodes and decouple the dependencies in SIT. Therefore, SCUE delivers high performance in NVM systems while ensuring root crash consistency.

VIII. Conclusion

In order to consistently and correctly update the root of the integrity tree with low overheads, this paper proposes the low-latency and shortcut updated scheme, called SCUE. The idea behind SCUE is that only the updates in persistent leaf nodes and on-chip root are necessary and sufficient to ensure the system integrity after system recovery. Propagating the modifications from leaf nodes to root incurs a long *crash window*. When crashes occur during the *crash window*, the root is inconsistent with leaf nodes. We instantaneously update the root in the SIT to remove the *crash window*. A counter-summing recovery approach is further proposed to decouple the dependencies in SIT, and provide the ability of SIT to be recovered from the consistent leaf nodes up after system reboots. Compared with state-of-the-art designs, the SCUE significantly reduces the system execution time while offering integrity verification after system failures and reboots.

Acknowledgements

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 62125202 and U22B2022 and Key Laboratory of Information Storage System, Ministry of Education of China.

References

[1] "A. m. rudo. 2016. deprecating the pcommit instruction." https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction.

[2] "eadr: New opportunities for persistent memory applications." https://software.intel.com/content/www/us/en/develop/articles/eadr-new-opportunities-for-persistent-memory-applications.html.

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

[4] M. Alwadi, A. Mohaisen, and A. Awad, "Promt: optimizing integrity tree updates for write-intensive pages in secure nvms," in *Proceedings of the ACM International Conference on Supercomputing*, 2021, pp. 479–490.

[5] M. Alwadi, K. Zubair, D. Mohaisen, and A. Awad, "Phoenix: Towards ultra-low overhead, recoverable, and persistently secure nvm," *IEEE Transactions on Dependable and Secure Computing*, 2020.

[6] 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.

[7] A. Awad, S. Suboh, M. Ye, K. A. 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)*. IEEE, 2019, pp. 610–614.

[8] 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*. ACM, 2019, pp. 104–115.

[9] M. Bellare, R. Canetti, and H. Krawczyk, "Message authentication using hash functions: The hmac construction," *RSA Laboratories' CryptoBytes*, vol. 2, no. 1, pp. 12–15, 1996.

[10] M. Bellare, D. Pointcheval, and P. Rogaway, "Authenticated key exchange secure against dictionary attacks," in *International conference on the theory and applications of cryptographic techniques*. Springer, 2000, pp. 139–155.

[11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, and S. Sardashti, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[12] 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, pp. 177–188.

[13] 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.

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

[15] Z. Dang, S. He, P. Hong, Z. Li, X. Zhang, X.-H. Sun, and G. Chen, "Nvalloc: rethinking heap metadata management in persistent memory allocators," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 115–127.

[16] G. Dhiman, R. Ayoub, and T. Rosing, "Pdram: A hybrid pram and dram main memory system," in *2009 46th ACM/IEEE Design Automation Conference*. IEEE, 2009, pp. 664–669.

[17] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan, "Data tiering in heterogeneous memory systems," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016, pp. 1–16.

[18] A. Freij, S. Yuan, H. Zhou, and Y. Solihin, "Persist-level parallelism: Streamlining integrity tree updates for secure non-volatile memory," in *Proceedings of the 53nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2020.

[19] A. Freij, H. Zhou, and Y. Solihin, "Bonsai merkle forests: Efficiently achieving crash consistency in secure persistent memory," in *2021 54rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2021.

[20] B. Gassend, G. E. Suh, D. Clarke, M. Van Dijk, and S. Devadas, "Caches and hash trees for efficient memory integrity verification," in *The Ninth International Symposium on High-Performance Computer Architecture, HPCA-9. Proceedings.* IEEE, 2003, pp. 295–306.

[21] S. Gueron, S. Johnson, and J. Walker, "Sha-512/256," in *2011 Eighth International Conference on Information Technology: New Generations*. IEEE, 2011, pp. 354–358.

[22] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[23] J. Huang and Y. Hua, "A write-friendly and fast-recovery scheme for security metadata in non-volatile memories," in *The 27th IEEE International Symposium on High-Performance Computer Architecture (HPCA-27)*, 2021.

[24] M. Imran, T. Kwon, and J.-S. Yang, "Adapt: A write disturbance-aware programming technique for scaled phase change memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 4, pp. 950–963, 2021.

[25] J. Jeong, J. Hong, S. Maeng, C. Jung, and Y. Kwon, "Unbounded hardware transactional memory for a hybrid dram/nvm memory system," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 525–538.

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

[27] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated persist ordering," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016, p. 58.

[28] D. Koo, Y. Shin, J. Yun, and J. Hur, "Improving security and reliability in merkle tree-based online data authentication with leakage resilience," *Applied Sciences*, vol. 8, no. 12, p. 2532, 2018.

[29] H. Lee, Y. Chen, P. Chen, P. Gu, Y. Hsu, S. Wang, W. Liu, C. Tsai, S. Sheu, and P. Chiang, "Evidence and solution of over-reset problem for hfo x based resistive memory with sub-ns switching speed and high endurance," in *International Electron Devices Meeting*, 2010.

[30] M.-J. Lee, C. B. Lee, D. Lee, S. R. Lee, M. Chang, J. H. Hur, Y.-B. Kim, C.-J. Kim, D. H. Seo, and S. Seo, "A fast, high-endurance and scalable non-volatile memory device made from asymmetric ta 2 o 5- x/tao 2- x bilayer structures," *Nature materials*, vol. 10, no. 8, p. 625, 2011.

[31] M. Lei, F. Wang, D. Feng, F. Li, and J. Xu, "An efficient persistency and recovery mechanism for sgx-style integrity tree in secure nvm," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 702–707.

[32] G. Liu, K. Li, Z. Xiao, and R. Wang, "Ps-oram: efficient crash consistency support for oblivious ram on nvm," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 188–203.

[33] S. Liu, A. Kolli, J. Ren, and S. Khan, "Crash consistency in encrypted non-volatile main memory systems," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 310–323.

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

[35] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling efficient and scalable hybrid memories using fine-granularity dram cache management," *IEEE Computer Architecture Letters*, vol. 11, no. 2, pp. 61–64, 2012.

[36] H. E. Michail, A. P. Kakarountas, A. Milidonis, and C. E. Goutis, "Efficient implementation of the keyed-hash message authentication code (hmac) using the sha-1 hash function," in *Proceedings of the 2004 11th IEEE International Conference on Electronics, Circuits and Systems, 2004. ICECS 2004.* IEEE, 2004, pp. 567–570.

[37] D. Naor, A. Shenhav, and A. Wool, "One-time signatures revisited: Practical fast signatures using fractal merkle tree traversal," in *2006 IEEE 24th Convention of Electrical & Electronics Engineers in Israel*. IEEE, 2006, pp. 255–259.

[38] H. Park, S. Yoo, and S. Lee, "Power management of hybrid dram/pram-based main memory," in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2011, pp. 59–64.

[39] M. Poremba, T. Zhang, and Y. Xie, "Nvmain 2.0: A user-friendly memory simulator to model (non-) volatile memory systems," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 140–143, 2015.

[40] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of pcm-based main

memory with start-gap wear leveling," in *Proceedings of the 42nd annual IEEE/ACM international symposium on microarchitecture*. ACM, 2009, pp. 14–23.

[41] J. Rakshit and K. Mohanram, "Assure: Authentication scheme for secure energy efficient non-volatile memories," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[42] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Proceedings of the international conference on Supercomputing*, 2011, pp. 85–95.

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

[44] 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.

[45] 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 *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 416–427.

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

[47] G. E. Suh, D. Clarke, B. Gasend, M. Van Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* IEEE, 2003, pp. 339–350.

[48] Y. Sung-Ming and L. Kuo-Hong, "Shared authentication token secure against replay and weak key attacks," *Information Processing Letters*, vol. 62, no. 2, pp. 77–80, 1997.

[49] S. Swami, J. Rakshit, and K. Mohanram, "Secret: Smartly encrypted energy efficient non-volatile memories," in *53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2016, pp. 1–6.

[50] M. Taassori, A. Shafiee, and R. Balasubramonian, "Vault: Reducing paging overheads in sgx with efficient integrity verification structures," in *ACM SIGPLAN Notices*, vol. 53, no. 2. ACM, 2018, pp. 665–678.

[51] J. M. Turner, "The keyed-hash message authentication code (hmac)," *Federal Information Processing Standards Publication*, vol. 198, no. 1, 2008.

[52] X. Wei, D. Feng, W. Tong, J. Liu, and L. Ye, "Morlog: Morphable hardware logging for atomic persistence in non-volatile main memory," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, pp. 610–623.

[53] F. Xia, D. Jiang, J. Xiong, and N. Sun, "Hikv: A hybrid index key-value store for dram-nvm memory systems," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 349–362.

[54] X. Xin, Y. Guo, Y. Zhang, and J. Yang, "Sam: Accelerating strided memory accesses," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 324–336.

[55] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the challenges of crossbar resistive memory architectures," in *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 476–488.

[56] 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.

[57] M. Ye, C. Hughes, and A. Awad, "Osiris: A low-cost mechanism to enable restoration of secure non-volatile memories." in *MICRO*, 2018, pp. 403–415.

[58] 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.

[59] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *ACM SIGARCH computer architecture news*, vol. 37, no. 3. ACM, 2009, pp. 14–23.

[60] K. A. Zubair and A. Awad, "Anubis: ultra-low overhead and recovery time for secure non-volatile memories," in *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 2019, pp. 157–168.

[61] P. Zuo, Y. Hua, and Y. Xie, "Supermem: Enabling application-transparent secure persistent memory with low overheads," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[62] P. Zuo, Y. Hua, M. Zhao, W. Zhou, and Y. Guo, "Improving the performance and endurance of encrypted non-volatile main memory through deduplicating writes," in *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 442–454.