A High-Performance and Fast-Recovery Scheme for Secure Non-Volatile Memory Systems

Yujie Shi, Yu Hua*, Jianming Huang
Wuhan National Laboratory for Optoelectronics, School of Computer
Huazhong University of Science and Technology, Wuhan, China
*Corresponding author: Yu Hua
E-mail: {shiyj, csyhua, jmhuang}@hust.edu.cn

Abstract—**Non-Volatile Memory (NVM) delivers high performance due to efficiently coalescing the properties of memory and storage while suffering from high security risks. To protect data, NVM systems introduce secure mechanisms, e.g., counter mode encryption and integrity verification, which necessitate extra security metadata. Unfortunately, these metadata are possibly lost when the system crashes, making data inaccessible. To ensure normal system operation after reboot, it is critical to recover these security metadata. The SGX-style integrity tree (SIT) can provide integrity verification for user data with high performance. Unfortunately, it is challenging for SIT to recover and verify the stale tree nodes due to the complex inter-layer dependency and the root inconsistency problem.**

In this paper, we propose Steins, to bridge the gap between fast recovery and high performance in secure NVM systems. Based on the consistency between the lost node and their persistent child nodes, Steins proposes an efficient counter generation scheme to make SIT recoverable. Moreover, Steins utilizes the offset-based tracking mechanism to locate the stale nodes during recovery. More importantly, to prevent attacks during recovery, Steins introduces efficient trust bases for verification. The evaluation results show that compared with state-of-the-art recovery schemes, Steins shows high runtime performance and short recovery time, while supporting low metadata storage overhead.

Index Terms—**Non-Volatile memory, memory security, integrity verification, metadata recovery**

I. Introduction

Non-volatile memory (NVM) has received significant attention due to its salient features, such as large capacity, non-volatility, byte-addressability, and low latency [1]–[3]. Consequently, numerous schemes utilize NVM (e.g., Intel's Optane DC Persistent Memory [4]) in HPC clusters to improve performance [5]–[8]. Unfortunately, NVM also suffers from limited write endurance, read-write asymmetry, and data security problems. Owing to its non-volatile nature, NVM systems require additional security mechanisms to ensure data confidentiality and integrity [9]–[11]. Traditionally, existing secure NVM systems adopt data encryption and integrity verification mechanisms, which introduce additional security metadata to protect user data. To reduce NVM access, these metadata are cached in the memory controller, which may be lost if the system crashes. To ensure the normal operation of the system, it is crucial to quickly recover stale metadata after the system crashes.

To safeguard data, the state-of-the-art schemes [9], [12]–[14] employ counter mode encryption (CME) [15] and integrity tree [10], [16] in secure NVM systems. Instead of directly encrypting data, CME employs AES [17] to generate the one-time pad (OTP) with a corresponding counter and data address as inputs and then XORs the data block with the OTP for encryption. During read operations, OTP generation and data retrieval from NVM occur simultaneously, effectively hiding the decryption latency.

To detect unauthorized data tampering, the system calculates the keyed-Hash Message Authentication Code (HMAC) [18] and stores it in NVM along with data. Without access to the secret key, attackers are unable to generate the correct HMAC for altered data. Therefore, the tampering attacks can be detected by comparing the re-calculated HMAC with the stored HMAC. However, attackers can record and replay the data and HMAC to circumvent HMAC verification, i.e., replay attacks.

In order to prevent replay attacks, Bonsai Merkle Tree (BMT) [19] and SGX-style Integrity Tree (SIT) [20] are commonly employed. The SIT/BMT guarantees both data integrity and the integrity of counters in CME, which is crucial for correct encryption/decryption. Specifically, the SIT/BMT associates the user data with the counters via the HMAC and prevents the counter blocks from replay attacks. In BMT, the counter blocks are hashed to generate HMACs as the parent nodes, which are further iteratively hashed to generate the root. The root is stored in the on-chip non-volatile register and trusted in the existing threat model [9], [10], [16], [21]. Unlike BMT, the SIT node consists of self-increasing counters and one HMAC. The HMAC is calculated over the counter block, the node address, and one corresponding counter in its parent node. In this paper, each security metadata (i.e., counter block in CME and integrity tree node) is 64 bytes, matching the cache line granularity.

Since the parent node calculation requires the HMAC of child nodes as input, the BMT needs to sequentially update the HMACs in the same branches when modifying leaf nodes. By employing the self-increasing counters as inputs, SIT enables parallel computation of HMACs of nodes at different levels, thus achieving higher performance than BMT. Moreover, since the counter can be compressed, SIT contains fewer levels than BMT, resulting in less update overhead [22]. For high performance, we leverage the SIT to protect NVM systems like [21]–[23].

To effectively adopt SIT in secure NVM systems, it is crucial to address the crash inconsistency problem. To reduce NVM access and improve performance, the security metadata are cached in the metadata cache within the memory controller [10], [21], [24], [25]. When flushing user data or dirty nodes into NVM, the corresponding cached parent nodes are modified, rendering their counterparts in NVM stale. If the system crashes, the dirty metadata that have not been persisted are lost. After rebooting, the stale metadata stored in NVM cannot provide adequate security protections due to the inconsistency [14], [16]. Therefore, it is essential to recover the stale metadata into the latest and consistent state. Unfortunately, the recovery of stale nodes presents challenges due to complex inter-layer dependencies [21], [23]. Furthermore, during recovery, ensuring the integrity of retrieved nodes is also difficult due to the inconsistency between the tree root and the whole tree nodes [14], [21].

As the cluster size increases, the likelihood of cluster storage system crashes also increases. [26], [27]. To guarantee the high availability of the system, instant recovery is necessary [28]–[30]. Due to the low-latency requirements from real-world applications and high-availability systems, we need to bridge the gap between fast recovery and runtime performance.

Unfortunately, existing schemes [14], [16], [21] cannot achieve both high performance and fast recovery at the same time. To recover SIT nodes, ASIT [16] atomically persists the modification of dirty metadata, resulting in $2\times$ memory writes. For verification, both ASIT and STAR [21] maintain an additional Merkle tree (cache-tree) based on cache/dirty nodes, leading to high computational costs. Besides, STAR also needs to sort the dirty nodes for updating the cache-tree. SCUE [14] only maintains the Recovery_root to verify the leaf nodes, thus achieving high performance. Unfortunately, SCUE needs to reconstruct the entire tree from all the leaf nodes during recovery, which requires hours for TB memory [16]. Furthermore, these schemes do not employ the split counter block in the leaf nodes, incurring large storage overhead.

In this paper, we propose cost-efficient Steins, a novel scheme supporting the fast recovery of SIT while guaranteeing high performance. Steins proposes a new counter-generation scheme to enable the restoration of stale SIT nodes from persisted child nodes. To accelerate recovery, Steins tracks the nodes required for recovery. Specifically, when a clean node becomes dirty, Steins writes its offset in the metadata region to the record lines within asynchronous DRAM refresh (ADR) [31] in the memory controller. Even if the system crashes, the record lines can be flushed into NVM with the extra power provided ADR [31]. To verify the retrieved nodes, based on the monotonic nature of the counters and the consistency between the root and its persistent child nodes, Steins introduces the proper trust bases with negligible maintenance overhead. Specifically, Steins maintains the total increases of cached counters of dirty nodes over their counterparts in NVM for each level, i.e., $LInc$s, to recover and verify the stale nodes from the root to the leaf.

To evaluate our proposed scheme, we use Gem5 [32] with NVMain [33] to implement Steins. To effectively demonstrate the performance, we evaluate Steins with typical persistent workloads and benchmarks from the SPEC2006 [34] and SPEC2017 [35], which have been widely used [14], [23]. The results show that Steins have higher performance than the existing schemes in all workloads. Additionally, Steins requires approximately 0.44 seconds to recover all the stale metadata in systems equipped with a 4MB metadata cache. In summary, our contributions are summarized as follows:

- **Efficient counter generation scheme for recovery.** To enable node recovery, we generate the parent counter from the child node during runtime. By adopting efficient monotonically increasing functions, we eliminate complex inter-layer dependency in SIT without sacrificing security.
- **Offset-based tracking for fast location.** We track the dirty nodes during runtime with low write and update overheads. Given the limited size of the metadata region, we utilize offset, which are smaller in size compared to addresses, to locate the node within the metadata region.
- **Appropriate trust bases for fast verification.** We leverage the root and the $LInc$ of each level to verify the retrieved nodes without involving irrelevant clean nodes. More importantly, the $LInc$ can be updated with negligible overhead, and only 8 bytes or less are required for its storage.
- **Extensive experiments and analysis.** We have implemented and evaluated Steins using representative benchmarks. Extensive experimental results show that our Steins achieves high performance and fast recovery while guaranteeing low storage overhead. Compared with the ASIT and STAR, our Steins improves the system performance by 20.7% and 12.7% respectively.

II. Background

A. Threat Model

It is well-recognized that only the on-chip processor domain is considered secure in the existing threat model [10], [12], [14], [21], [36]. An attacker can obtain the memory data in several ways, such as snooping the memory bus and scanning memory, which violate data confidentiality [16], [21]. In addition, the attacker can tamper with or replay the in-memory data by attacking the memory or memory bus, which violates data integrity. Our work focuses on both data confidentiality and integrity. Other attacks, such as the Meltdown [37] and Spectre [38], are beyond the scope of this paper.

B. Data Confidentiality

To prevent data leakage, it is necessary to encrypt data when flushing it into NVM. However, if the system directly encrypts/decrypts data, the read performance significantly decreases since the decryption latency is comparable to NVM read latency. Besides, the same data block will be encrypted into the same ciphertext, thus the dictionary attack cannot be avoided. To mitigate these issues, Counter Mode Encryption (CME) is commonly employed in current systems [15], [16].

When flushing data, CME utilizes a secret key, a counter, and the data address to create the one-time padding (OTP) via

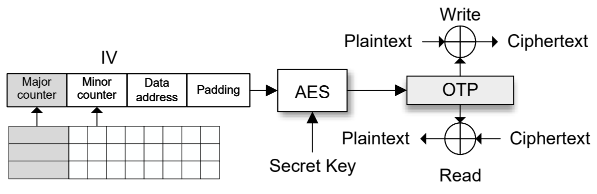

Fig. 1. The Counter Mode Encryption.

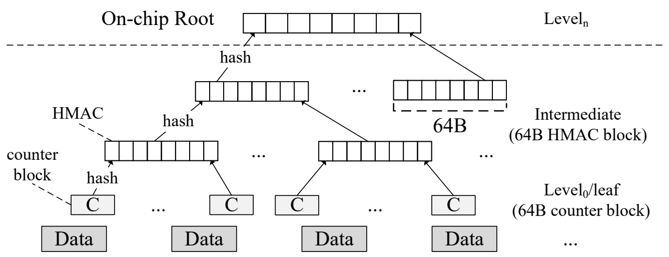
On-chip Root
Level$_n$
hash
HMAC
hash
64B
Intermediate (64B HMAC block)
counter block
hash
C
Level$_0$/leaf (64B counter block)
Data

Fig. 2. The Bonsai Merkle Tree.

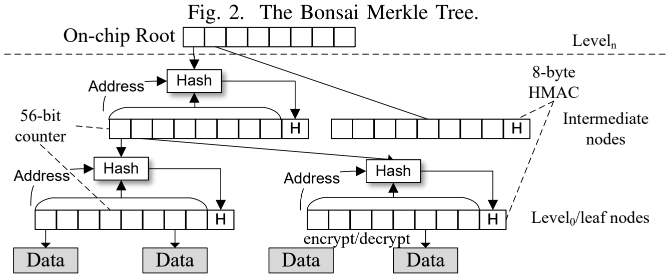

Fig. 3. The SGX-style Integrity Tree.

AES. The data is then XORed with this OTP for encryption. While fetching data from NVM, the system re-generates the OTP with the counter blocks cached in the memory controller in parallel, thus hiding the decryption latency. Finally, the original data is obtained by XORing the ciphertext with the OTP.

In CME, the OTP is not reused because the inputs for each write are unique. For different data blocks, their addresses are distinct. For the same data blocks, each write operation increments the corresponding counter by one. Therefore, CME can provide higher confidentiality than direct encryption.

In general, a 64B counter block in CME consists of 8 64-bit counters, covering 8 data blocks. To reduce metadata storage overhead, existing works [16], [39], [40] widely adopt the split counter scheme [41]. As shown in Fig. 1, One split counter block consists of one 64-bit major counter and 64 7-bit minor counters, thus covering 64 data blocks. When flushing data, the minor counter is utilized in conjunction with the major counter to generate the OTP. When the minor counter overflows, to guarantee the uniqueness of OTP, all the minor counters are reset to zero, the major counter increases by one, and the corresponding data blocks are re-encrypted. Since the major counter size is 64 bits, it is hard to overflow in the lifespan of NVM [14], [40].

C. Data Integrity

To detect unauthorized modifications, i.e., tampering attacks, existing schemes employ HMAC for data protection [16], [21], [42]. The system generates an HMAC from the data, its address, and a secret key during data flush operations, storing it alongside the data. Upon retrieval, the system recalculates the HMAC and compares it against the stored version. Without the secret key, attackers cannot compute the correct HMACs. Therefore, any discrepancy between the regenerated and stored HMACs indicates a potential attack.

However, attackers can record the data and corresponding HMAC via attacking memory bus or NVM [16], [21]. If attackers replace new data and HMAC with the old data and HMAC, the system cannot recognize this attack via checking HMAC. To identify these replay attacks, the system uses integrity trees, such as Merkle Tree (MT) [43], Bonsai Merkle Tree(BMT) [19], and SGX-style Integrity Tree (SIT) [20], to verify data.

To guarantee correct data encryption, the BMT combines CME and MT. As shown in Fig. 2, the BMT leaf nodes are the encrypted counter blocks in CME, which are hashed to generate HMAC blocks at the upper level, i.e., parent nodes. These parent nodes are recursively hashed up to the root node, which is stored on-chip and considered invulnerable in our threat model. If attackers attempt to replay a data block, they must also replay all corresponding branch nodes, including the invulnerable root, which is infeasible. Since the calculation in the parent node requires its child's HMAC as an input, BMT necessitates sequential HMAC calculations along the branch, which incurs large computation overhead.

Unlike BMT, the SIT enables parallel updates of HMACs across nodes at different levels [20]. As shown in Fig. 3, the SIT node consists of one 64-bit HMAC and eight 56-bit counters, one for each child node. The HMAC in SIT is calculated over the counter block, the address, the corresponding parent counter, and the secret key. By employing self-increasing counters that can updated independently, SIT can compute HMACs of different levels in parallel, thus achieving higher performance than BMT. Additionally, the security model of SIT is superior to that of BMT as each counter is doubly protected by the HMAC in both its node and its child node [21]. Moreover, due to the compressibility of counters, SIT encompasses fewer levels than BMT, which further optimizes its structure [22].

In this paper, each security metadata (i.e., encrypted counter block and tree node) is 64 bytes, matching the cache line granularity. To improve performance and reduce write traffic, these security metadata are cached within the memory controller. During runtime, the cached nodes are trusted since they are verified before entering the cache. In SIT, when reading the tree node into the metadata cache, the system re-calculates the HMAC with its parent counter as input and compares it with the stored HMAC. If the parent node is not cached, it is fetched and verified by its own parent counter. If cache misses occur again, this process continues recursively up the tree, until hitting the on-chip root.

There are two options to update the SIT tree, including the eager update and the lazy update scheme. In the eager update scheme, when evicting a dirty data block, the system needs to update all ancestor nodes in the corresponding branch. This modification process possibly incurs significant memory access and computation overhead if cache misses. In the lazy update scheme, when evicting a dirty data block/tree node into NVM, only its parent node needs to be cached and updated. The other ancestor nodes in the same branch remain unchanged unless their child nodes are evicted.

D. Motivation

To enhance performance and minimize memory writes, we employ SIT with the lazy update scheme to secure NVM systems. If the system crashes, the dirty metadata that have not been persisted are lost. After rebooting, the stale metadata in NVM fails to provide a security guarantee due to the inconsistency. To operate properly after rebooting, the system must correctly restore the stale metadata to the latest consistent state while preventing integrity attacks.

For BMT, the tree can be reconstructed from leaf nodes since the nodes can be calculated from child nodes. However, due to the complex dependency, recovering stale nodes in SIT from persistent child nodes is challenging. In SIT, recovering a node requires its parent counter as input, which possibly also needs to be recovered. Besides, with the lazy update scheme, the node modification only propagates to their parent nodes. Therefore, the tree root fails to reflect the latest states of all SIT nodes. As a result, even if we can recover the stale nodes, the retrieved nodes cannot be verified by the SIT. In other words, the attacks during recovery cannot be detected.

As the cluster size increases, the system crashes become frequent [27]. For the high-availability systems that need to meet the “five nines“ rule, instant recovery is important [28], [29]. Besides, for high-performance systems, the extra mechanisms introduced for fast recovery and verification should keep minimal overhead [16], [21]. Unfortunately, existing state-of-the-art schemes cannot achieve fast recovery and high performance at the same time. Existing works, including Osiris [44], Supermem [40], and SCA [45], ensure that counter blocks are consistent with or recovered from user data. However, they cannot be used to recover the SIT nodes.

To recover the SIT nodes, ASIT [16] atomically persists the updated metadata blocks to the Shadow Table (ST), incurring $2\times$ memory writes and large performance overheads. Besides, for verification, ASIT maintains an extra Merkle tree (i.e. cache-tree) based on cached nodes. Once the cached nodes are modified, the modifications are propagated into the cache-tree root through multiple HMAC calculations, leading to expensive computation overhead and long latency.

Similar to ASIT, STAR [21] also maintains the cache-tree based on the dirty nodes. When updating cache-tree, STAR needs to sort the dirty nodes in the same set by the addresses to calculate the set-MAC, incurring extra overhead. Moreover, STAR leverages the multi-layer bitmap to mark the dirty nodes. The bitmap is updated when the state of nodes changes

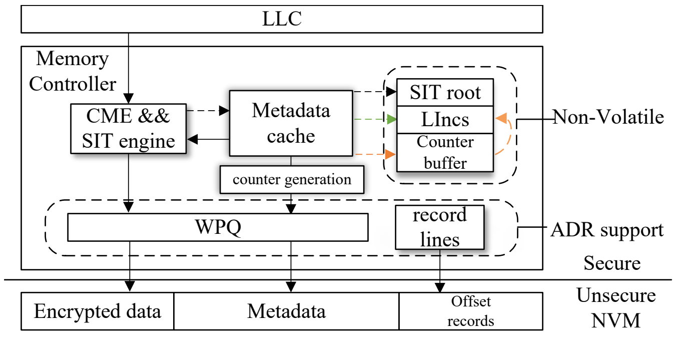

Fig. 4. The overview of Steins.

from clean to dirty or vice versa, leading to extra memory access overhead.

SCUE [14] maintains the Recovery_root which is the sum of all the leaf counters. During recovery, SCUE reconstructs the whole tree from leaf nodes by summing the counters and compares the retrieved root with the Recovery_root for verification. Even if SCUE achieves high performance, the recovery time is hour-scale for TB memory, which is unacceptable.

To balance performance and recovery, we introduce a cost-effective approach named Steins, which enables fast recovery with limited writes and negligible update overhead. To reduce storage overhead, Steins employs the split counter block in the SIT leaf nodes. To match the cache line granularity, in SIT, the major counter is set to 64-bit and the minor counter is set to 6-bit. For leaf nodes, we store the major counter in the HMAC of the data block for recovery. When the major counter overflows, we use the write-through scheme [40] to persist the counter blocks, avoiding the major counter recovery.

III. SYSTEM DESIGN

A. Steins Overview

To achieve fast recovery with negligible overhead, our Steins proposes an efficient counter generation scheme for node recovery, tracks the dirty nodes for fast location, and introduces the new trust bases for verification. Fig. 4 shows the overview architecture of Steins.

We observe that in the lazy update scheme, the cached nodes remain consistent with their persistent child nodes, providing an opportunity for recovery. Based on this, Steins decouples the complex inter-layer dependency to enable node recovery. As shown in Fig. 5, instead of using a self-increasing counter, Steins generates the parent counter from the child node via the pre-defined functions during runtime. Therefore, during recovery, Steins can recover the stale counters from persistent child nodes. Further, the HMAC can also be re-generated with the parent counter calculated from the retrieved counters.

As only dirty nodes need to be recovered, rapidly locating these nodes can accelerate the recovery process. To this end, Steins employs an offset-based tracking scheme to record the dirty nodes. Additionally, Steins caches partial record lines in the memory controller to decrease NVM access and utilizes ADR to ensure crash consistency.

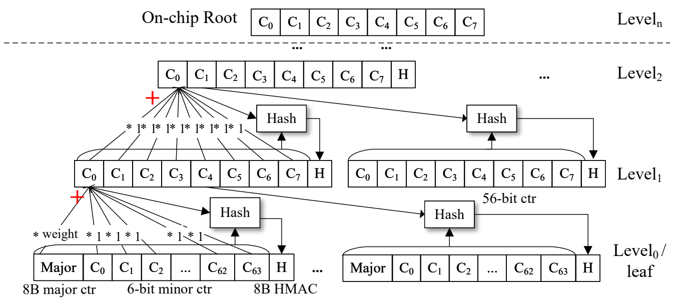

Fig. 5. The reconstructed SIT in Steins.

More importantly, Steins introduces the trust bases, i.e., L_kIncs ($0 \leq k \leq n-1$, the root located in $Level_n$), to verify the retrieved nodes. The L_kInc quantifies the total increase in the cached counters of $Level_k$ nodes compared to their stale versions in NVM. Compared to the cache tree, the L_kIncs can be updated efficiently. Moreover, Steins eliminates the iterative reads of ancestor nodes from the write critical path by using a small non-volatile buffer to temporarily store the parent counter. During recovery, after locating the dirty nodes according to offset records, Steins recovers and verifies the stale nodes from the higher level to the lower level.

B. Counter-Generation Scheme for Recovery

With the lazy update scheme, the cached SIT node is modified only when its child nodes are flushed into NVM. Therefore, the dirty nodes are consistent with their persistent child nodes. Thus, we consider how to recover the stale nodes from the child nodes stored in NVM. In general, an SIT node consists of one counter block and one HMAC, which is calculated over the counter block and its corresponding parent counter. Therefore, if the parent counter can be generated from the child nodes, the stale node can be successfully restored.

We observe that as long as the counter block is not reused, the SIT can guarantee data integrity. Due to the existence of HMAC, attackers cannot successfully tamper with the node without the secret key. Besides, during runtime, the modifications of flushed nodes will be propagated to the cached nodes/root, which is invulnerable in our threat model. If attackers attempt to replay stale nodes, the recalculated HMAC using the new counter as input will not match the stored HMAC, signaling an unauthorized replay attack.

Based on this observation, while ensuring that the counter is monotonically increasing, Steins proposes an efficient counter generation scheme to achieve the recoverable SIT.

1) Counter generation scheme: As shown in Fig. 5, when flushing a dirty node, instead of using the self-increasing counter, Steins generates the parent counter from the evicted nodes via the monotonically increasing linear function.

In the general node, the counter block consists of eight similar 56-bit counters. To efficiently calculate the parent counters of these nodes, Steins adopts the sum function directly as shown in Equation (1).

$$Parent = C_0 + C_1 + C_2 + ...C_6 + C_7 \tag{1}$$

Unfortunately, the simple sum function is not suitable for the leaf nodes that employ split counter block. In the split counter block, when a minor counter overflows, all of the minor counters are reset to zero, and the major counter increases by 1. To maintain the monotonically increasing property of the generated parent counter, it is essential to treat the major and minor counters separately. For efficient computation, Steins employs a linear function that assigns different weights to the major and minor counters, as detailed in Equation (2).

$$Parent = Major * weight + C_0 + C_1 + \ldots + C_{62} + C_{63} \tag{2}$$

An intuitive generation scheme is to assign a weight of 1 to each minor counter and assign the weight of the maximum sum of the minor counters (i.e., $2^6 * 64$) to the major counter. In this case, even if all the minor counters are reset to zero when one minor counter overflows, the parent counter still increases. Unfortunately, the overflow probability of the parent counter increases significantly, since the actual sum of the minor counters is generally smaller than the maximum [42].

To reduce the overflow possibility while keeping the counter monotonically increasing, Steins updates the major counter by skipping. Specifically, Steins assigns the weight of the maximum of the minor counter (i.e., 2^6) to the major counter. When one minor counter overflows, rather than increasing the major counter by one, Steins calculates the sum of the minor counters, divides it by the maximum value of the minor counter (i.e., 2^6), rounds up to obtain the increment, and updates the major counter by adding this increment. In this way, when the minor counter overflows, the parent counter is aligned upwards in multiples of the maximum of the minor counter (i.e., 2^6), thus increasing monotonically.

2) Overflow analysis: Traditionally, the 56-bit counter in the SIT tree node counts the number of the total memory writes for the data that the counter covers and does not overflow during the lifetime of NVM [14]. Assuming that the memory write latency is 300ns [14], [21], [40], the system requires $2^{56}\times$ 300ns (about 685 years) to overflow the 56-bit counter. If Steins employs the split counter scheme, in the corner cases where the sum of minor counters reaches $2^6 + 1$ (immediately following a minor counter overflow), the major counter is increased by two. As a result, the parent counter corresponds to twice the number of memory writes compared to the traditional SIT model. Therefore, the 56-bit counter would require at least 342 years to overflow [21]. Even in the worst cases, i.e., the major counter overflows, the system can change the secret key and reconstruct the tree for integrity protection.

Since both predefined functions are much simpler to calculate compared to HMAC, the parallelism of SIT is only slightly affected. During recovery, Steins recovers the counters of stale nodes from persistent child nodes via predefined functions.

C. Offset-based Tracking Mechanism

For TB-scale NVM, recovering all the SIT nodes will consume several hours, which is unacceptable for the high-availability system [21], [23]. To accelerate the recovery process, it is necessary to locate the nodes that need to be recovered. As only the cached dirty nodes are lost during crashes, Steins tracks these dirty nodes during runtime. Specifically, Steins maintains a record for each metadata cache line in NVM to store the dirty node location of the cache.

Given that the metadata region in NVM only constitutes a minor portion, employing 8B addresses for tracking is unnecessary. Therefore, Steins utilizes the offset of a node in the metadata region to locate the node. The offset size is dictated by the metadata region's size. In this paper, we set the offset to 4 bytes, which enables a metadata area of up to 256GB and is sufficient for most systems. In this case, a 64B record line can cover 16 nodes. For the 256KB metadata cache, Steins allocates the 16KB record region in NVM.

To minimize NVM access, Steins caches partial record lines in the memory controller and manages these lines using the LRU strategy. Meanwhile, Steins utilizes the ADR mechanism to ensure that these cached record lines can be flushed to NVM during crashes. When the cached node becomes dirty for the first time, Steins updates the record line based on the position of the dirty node in the cache. If the record line is not cached, Steins reads it from NVM into the ADR area for updates.

In our system, treating clean nodes as if they are dirty does not compromise verification correctness (as described in Section III-H). Consequently, when dirty nodes are flushed into NVM, i.e., becoming clean, Steins does not update the record lines. During recovery, Steins reads all the records from the NVM to identify the dirty nodes. Given the limited cache size, the additional overhead incurred by recovering clean nodes is minimal.

D. Appropriate Trust Bases for Recovery

During the recovery process, the attackers can modify the retrieved nodes by tampering with/replaying the child nodes and eventually compromise the system. Therefore, it is necessary to verify the retrieved nodes. However, with the lazy update scheme, the root possibly does not reflect the latest node states, disabling these verifications. To prevent integrity attacks during recovery, we need to introduce extra trust bases for recovery verification, i.e., RecoveryTB.

In order to ensure fast verification and maintain high performance during runtime, the RecoveryTB needs to meet the following requirements: 1) the RecoveryTB can verify the retrieved nodes with as few clean nodes as possible involved. 2) the RecoveryTB can be updated easily and quickly to be consistent with modified nodes. 3) the RecoveryTB is small enough to store on-chip non-volatile regions.

By analyzing the SIT with a lazy update scheme intensively, we obtain the following observations: 1) The root can verify its child nodes stored in NVM, which is guaranteed by the consistency between the on-chip node and its persistent child nodes [14], [21]. 2) Due to the existence of HMAC, attackers are limited to replay attacks [14], [21]. 3) Due to the monotonically increasing property, the retrieved counters will be smaller than the latest version if replay attacks occur.

Based on these observations, we introduce the increments for each level. The $Level_k$ increment, denoted as L_kInc, means the total increase in the cached counters of $Level_k$ nodes compared to their stale counterparts stored in NVM. Since the cached clean node is the same as its counterpart in NVM, the L_kInc also represents the total increase in counters of $Level_k$ dirty nodes compared to their stale versions in NVM. By using $LIncs$ as RecoveryTB, Steins can verify the retrieved nodes from root to leaf without involving unnecessary clean nodes.

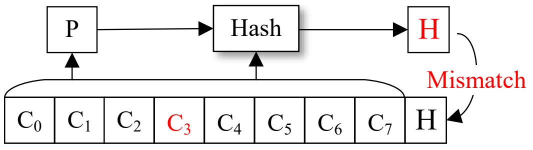

Fig. 6. The verification for each child node.

During recovery, the integrity attacks on SIT nodes can be categorized into tampering attacks and replay attacks, which are detected by HMAC and RecoveryTB respectively. As shown in Fig. 6, during recovery, the child nodes used for recovery can be verified by the HMAC with the retrieved counter as input. If the counter is modified, the calculated HMAC will mismatch the stored HMAC. As a result, the attackers are limited to replay attacks.

To prevent replay attacks, Steins recovers the SIT from the higher level to the lower level. When recovering $Level_k$ nodes, Steins will recalculate the total difference between the retrieved counters and their stale counters stored in NVM and then compare it with L_kInc. For $Level_{n-1}$, the attackers cannot attack the stale counters stored in NVM successfully as they can be directly verified by root. If the attackers replay the child nodes, the retrieved counters will be smaller than the correct counters. As a result, the recalculated $L_{n-1}Inc$ will mismatch the stored $L_{n-1}Inc$, indicating an attack. If they are matched, the $Level_{n-1}$ stale nodes are recovered successfully and can be used to verify $Level_{n-2}$ nodes. Finally, all the stale nodes can be recovered and verified in a similar way.

Compared to the cache-tree, the $LInc$s can be updated efficiently (described in Section III-E). An 8-byte size is sufficient for L_kInc to record the total increase in $Level_{k-1}$ cache nodes compared to their persistent versions. Therefore, a 64B non-volatile register can store all eight $LInc$s, which is enough for 16GB memory with 9-level SIT (including the root).

E. Flushing Metadata Efficiently

During runtime, when evicting a $Level_k$ dirty node due to cache replacement, the $Level_{k+1}$ parent node is updated with the newly generated counter. Correspondingly, the modifications propagate to both the L_kInc and $L_{k+1}Inc$. Specifically, the L_kInc is decreased by the increment between the evicted node's counters and their stale counterparts in NVM. The $L_{k+1}Inc$ is increased by the difference between the newly generated parent counter and the original parent counter. Actually, these two increments are equal, as the original parent counter is the sum of the counters of the persistent child

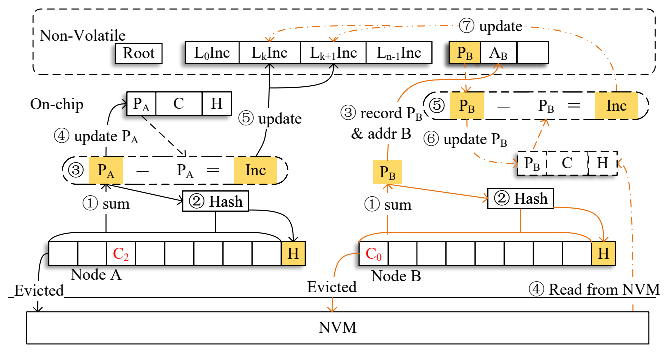

Fig. 7. The process of flushing dirty nodes.

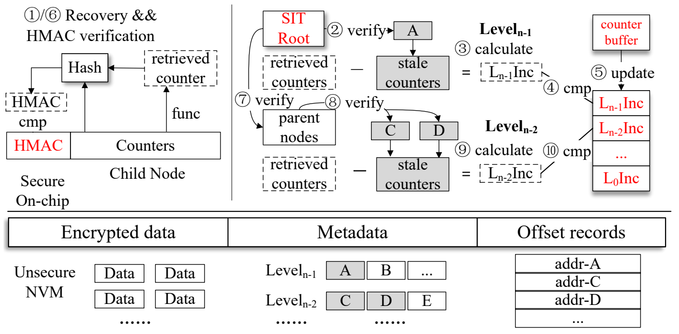

Fig. 8. The recovery process.

node [1]. Therefore, it is not necessary to read the stale node from NVM. Steins can calculate the increment by subtracting the newly generated parent counter from the original parent counter.

If the parent node is the root or already cached, the RecoveryTB can be updated directly. However, if the cache misses, the system needs to read the parent node, which may incur iterative reads for verification. In fact, in existing works [16], [21], [46], the parent node is necessary for HMAC calculation, making this overhead inevitable. However, even without the parent node, Steins can calculate the HMAC with the generated counter from the evicted node.

To boost performance, we aim to remove parent node reads from the write critical path. However, reading the parent node after completing the write operation may lead to inconsistencies between the RecoveryTB and the SIT if crashes. To ensure crash consistency, Steins employs a non-volatile buffer to temporarily store the newly generated parent counter.

As shown in Fig. 7, when flushing dirty node A whose parent node is cached, Steins generates the parent counter (P_A) in step ① and calculates the HMAC with the generated counter as input in step ②. In step ③, Steins calculates the increment by subtracting the generated P_A from the original parent counter. After that, Steins updates the parent node with generated counter, $L_k Inc$ and $L_{k+1} Inc$ with the increment in steps ④ and ⑤.

If the dirty node B whose parent node is not cached is flushed, Steins also generates the parent counter (P_B) in step ① and HMAC in step ②. Instead of reading the parent node from NVM directly, Steins writes the generated parent counter and the address of node A to the non-volatile buffer in step ③. After that, the write operation is completed. To circumvent checks on the non-volatile buffer when reading nodes, Steins fetches the parent nodes from NVM in step ④ before performing the next read operation or when the buffer is full. Then, Steins generates the increment in step ⑤ and then updates the parent node and RecoveryTB in steps ⑥ and ⑦.

During recovery, Steins checks the non-volatile buffer to propagate the modifications to RecoveryTB. For example, assuming that the generated counter at $Level_{k+1}$ is stored on this buffer when recovering $Level_{k+1}$ nodes, Steins will calculate the increment by subtracting the generated counter and stale counter which can be verified by its parent and then updates $L_k Inc$ and $L_{k+1} Inc$.

[1]The sum of split counter block is calculated via Equation (2)

F. Data Read and Write Operation

When a dirty data block is evicted from the Last Level Cache (LLC), the corresponding counter in the leaf node increases by one to generate OTP. The data block is then encrypted by XORing with the OTP and the HMAC is calculated over the data block and the counter. The $L_0 Inc$ is then updated to be consistent with the latest nodes. Meanwhile, Steins records the location of this modified node for recovery if it was clean before. After that, the encrypted data block and corresponding HMAC can be flushed into NVM. If cache misses, the leaf node will be read from NVM and verified by its parent node. Similarly, when fetching data from NVM, the system generates the OTP with the corresponding counter in parallel for encryption. If the counter is not cached, the system also needs to fetch the corresponding counter block from NVM.

When reading metadata into the cache, the cache replacement occurs if the metadata cache is full. If a dirty tree node is evicted, Steins will update the parent node and RecoveryTB as described in Section III-E. Similarly, if the parent node is clean before, Steins tracks it for recovery.

G. Recovery from the Root to the Leaf

When the system reboots after crashes, Steins initially fetches offset records from NVM to identify nodes requiring recovery. As shown in Fig. 8, Steins recovers and verifies the stale nodes from the higher level to the lower level. Initially, it locates dirty nodes at $Level_{n-1}$ and retrieves stale counters from their persistent child nodes in step ①. Steins then verifies each child node by recalculating its HMAC using the retrieved counter as input and compares this with the stored HMAC. If the calculated HMAC matches the stored one, there are no tampering attacks.

Subsequently, Steins reads the $Level_{n-1}$ stale nodes (e.g., node A) from NVM and verifies them with the SIT root in step ②. If the verification is successful, the stale counters can be trusted. Steins then calculates the total increment of the retrieved counters compared to those stored in NVM and compares the result with $L_{n-1}Inc$ in steps ③ and ④. If replay attacks occur, the calculated $L_{n-1}Inc$ will be smaller than stored $L_{n-1}Inc$ due to the inherently monotonically increasing nature of the counter. Notably, if there are no dirty $Level_{n-1}$ nodes, Steins still compares the $L_{n-1}Inc$ with zero to detect the integrity attacks.

Following the recovery of $Level_{n-1}$ nodes, Steins proceeds to recover the $Level_{n-2}$ dirty nodes (e.g., nodes C and D). Initially, Steins checks the non-volatile buffer to update the $L_{n-2}Inc$ and $L_{n-3}Inc$ for consistency in step ⑤. Similarly, Steins recovers the $Level_{n-2}$ nodes in step ⑥. With the $Level_{n-1}$ nodes already retrieved, Steins can iteratively verify the stale versions of the stale $Level_{n-2}$ nodes using these retrieved nodes or the root in steps ⑦ and ⑧. Thus, Steins can verify the retrieved $Level_{n-2}$ nodes using $L_{n-2}Inc$ in steps ⑨ and ⑩. Ultimately, all the SIT stale nodes can be recovered and verified.

Notably, the dirty node is consistent with its persistent child nodes whether they are stale/clean. Steins just recovers the SIT nodes to the state before crashes. Meanwhile, all the retrieved nodes will be marked as dirty to guarantee that the modification can propagate to upper-level nodes.

H. Security Analysis

During runtime, since the counter block is never reused in the lifetime of NVM, the SIT in Steins can guarantee data integrity like existing schemes [16], [21]. During recovery, the integrity attacks on the SIT nodes can be detected by the HMAC and RecoveryTB as described in III-D.

Besides, attackers may alter the state of nodes (i.e., changing clean nodes to dirty or vice versa) by modifying the offset records. Fortunately, in our Steins, marking clean nodes as dirty won't affect the correctness of recovery. For clean nodes, since the retrieved counters are the same as their stale versions stored in NVM, the calculated increment is indeed zero. Therefore, marking the clean nodes as dirty does not change the calculated $LInc$. Indeed, our Steins does not modify the records when the dirty nodes become clean during runtime.

If dirty nodes are marked as clean, Steins will not calculate the increment for these nodes, which is more than zero. In this case, similar to the result of the replay attacks, the recalculated $LInc$ will be smaller than stored $LInc$, signaling a potential attack. By leveraging these strategies and checks, Steins effectively mitigates the risks posed by both tampering and replay attacks, ensuring the correctness of the recovery process.

Similar to ASIT and STAR, Steins ensures integrity during system crashes. Moreover, Steins detects the attacked node levels via top-down verification, thus facilitating attack localization.

IV. Evaluation Methodology

Since hardware changes are challenging, we implement our Steins in simulation. For real processors, counter generation and $LInc$s updates need careful design for better performance.

TABLE I
THE CONFIGURATIONS OF THE EVALUATED NVM SYSTEM.

Processor	
CPU	8 cores, X86-64 processor, 2 GHz
Private L1i/d cache	32KB, 2-way, LRU, 64B Block
Shared L2 cache	512KB, 8-way, LRU, 64B Block
Shared L3 cache	2MB, 8-way, LRU, 64B Block
DDR-based NVM	
Capacity	16GB
PCM latency model	tRCD/tCL/tCWD/tFAW/tWTR/tWR =48/15/13/50/7.5/300 ns
Write queue	64 entries
Secure Parameters	
Metadata cache	256KB, 8-way, LRU, 64B Block
SIT	8/9 levels, 8-way, 64B Block
Hash latency	40 cycle
Non-volatile buffer	128B
offset records	16KB, 16 lines in memory controller

Moreover, the metadata cache sizes become limited by on-chip resources. To comprehensively evaluate Steins, we model Steins in the cycle-accurate Gem5 [32] and NVMain [33] simulator, which has been widely used in existing works [14], [40], [47], [48]. Table I shows the main configurations and parameters of Steins. We model a 16GB NVM and set the latency similar to existing designs [14], [21], [23], [39], [40]. Generally, the height of SIT is 9 (including the root). When using the split counter block in leaf nodes, the height of SIT is reduced by 1, i.e., 8. Since our Steins plays a role in memory access, in order to observe the performance impact of Steins, similar to existing studies [14], [23], we choose to use a 256KB metadata cache in the memory controller to store recently accessed metadata. In general, larger cache sizes deliver higher performance since the memory access decreases.

To effectively demonstrate and compare the performance of our new non-volatile memory system, we have selected eight representative benchmarks from SPEC2017 [35] and SPEC2006 [34] from ASIT, and implemented two persistent workloads from STAR to evaluate Steins. For each workload, we run 2 billion instructions after a 10 million instructions warm-up in GEM5.

To assess the effectiveness of Steins, we evaluate the subsequent schemes for comparisons. We does not compare our Steins with the SCUE, since it needs to reconstruct the whole tree, incurring unacceptable recovery time [14].

- **Anubis for SGX Integrity Tree scheme (ASIT).** ASIT persists the modifications of dirty metadata into the shadow table for recovery. In addition, ASIT constructs a 4-level cache-tree based on cached nodes for verification.
- **SIT trace and recovery scheme (STAR).** STAR stores the LSBs of the parent counter in the child node and tracks the dirty nodes using the bitmap for recovery. Similarly, STAR maintains a 4-level cache-tree based on dirty nodes. When updating cache-tree, STAR needs to sort the nodes in the same set by addresses.
- **Our proposed scheme (Steins-SC & Steins-GC).** Steins uses counter generation scheme to support recovery. By

recording the offset of dirty nodes, Steins locates the nodes that are required to be recovered. Meanwhile, Steins leverages the $LInc$s to verify the retrieved nodes. Since neither ASIT nor STAR considers the split counter block, to facilitate fair comparisons, we employ the general counter block and split counter block in the leaf nodes to achieve our Steins respectively, called Steins-GC and Steins-SC.

- **Write Back scheme (WB-SC & WB-GC).** WB uses the general CME and SIT in NVM systems to ensure data security. In WB, the system only writes modified metadata back to the NVM when the cache replacement occurs. We use the WB as the baseline, which does not support recovery. Similar to Steins, we implement the WB-SC and WB-GC respectively.

A. *Performance Analysis*

Fig. 9, Fig. 10 and Fig. 11 show the execution time, write and read latency of different schemes, respectively. We can observe that our Steins is better than STAR and ASIT in all workloads. On average, ASIT exhibits 1.20× execution time and 2.14× write latency compared with WB-GC. STAR obtains 1.12× the execution time and 1.67× write latency than WB-GC, outperforming Anubis. Our Steins-GC only increases write latency by 6% and even decreases read latency by 0.02%. The results show that in all the workloads, our Steins-GC has a similar performance to that of WB-GC while supporting fast recovery.

Compared to ASIT and STAR, our Steins-GC enhances system performance by 20.7% and 12.7%, respectively. This is because our Steins introduces minimal overhead and optimizes the metadata write operation. In Steins, the parent counter generation scheme uses simple linear functions, which only incur negligible overhead. Besides, Steins caches the record line in the memory controller and updates the record line only when the clean node is modified for the first time. When evicting the dirty node, Steins can calculate the HMAC even if its parent node is not cached. Thus, the iterative node reads are removed from the write critical path, which is unavoidable in other schemes including WB. ASIT and STAR require sequential cache-tree updates when modifying cached metadata, which Steins avoids. For memory-intensive workloads, frequent dirty node evictions allow Steins to achieve much better performance than Anubis and STAR, though Steins requires a little more resources for higher performance.

Fig. 12 shows that Steins-SC has a much better performance than Steins-GC. Compared to WB-SC, Steins-SC has 0.998× execution time on average. As the split counter block corresponds to 64 user data blocks, Steins-SC has a higher metadata cache hit rate and lower height of the SIT than Steins-GC (8 vs 9). The result shows that the Steins-SC reduces the execution time by 39% compared to Steins-GC. Therefore, it is necessary to support the recovery of the SIT with the split counter scheme.

In summary, integrating Steins into secure NVM systems introduces only a negligible overhead compared to the baseline without recovery support, significantly outperforming existing state-of-the-art schemes.

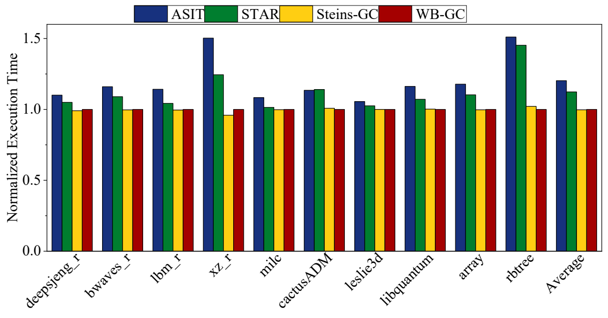

Fig. 9. The execution time (normalized to WB-GC).

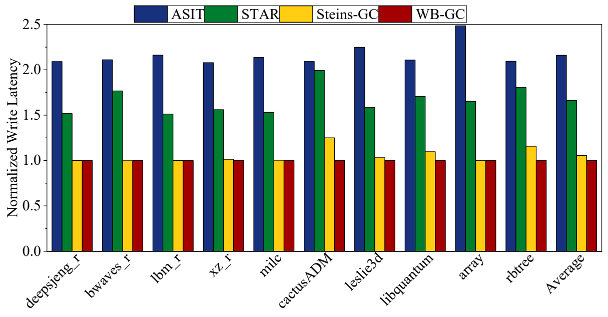

Fig. 10. The write latency (normalized to WB-GC).

B. *Write Traffic of different schemes*

Fig. 13 and Fig. 14 illustrate the write traffic of different schemes. When modifying the cached nodes, ASIT persists the shadow table blocks to the shadow table, causing 2× memory writes than WB-GC. When the state of the cached node changes from clean to dirty or vice versa, STAR needs to update the multi-layer bitmap lines cached within the memory controller, which may incur multiple memory access. On average, STAR incurs 1.3× memory traffic than WB-GC. Steins modifies the record lines only when the clean nodes become dirty, thus achieving lower memory access than STAR. On average, Steins-GC has 1.05× memory traffic than WB-GC. In both Steins and STAR, the memory traffic depends on the access pattern of workloads. The workloads with random

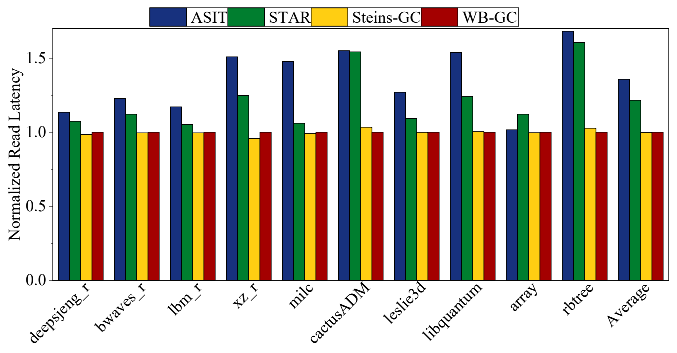

Fig. 11. The read latency (normalized to WB-GC).

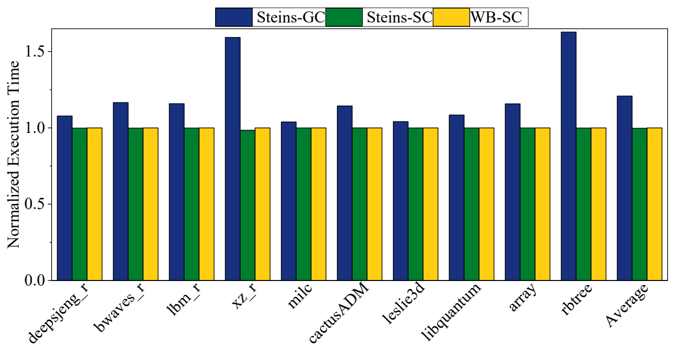

Fig. 12. The execution time (normalized to WB-SC).

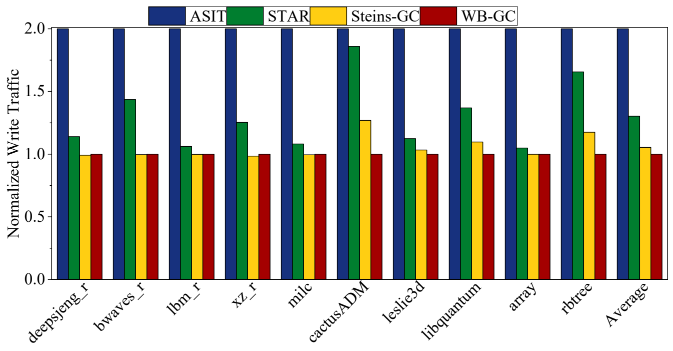

Fig. 13. The write traffic (normalized to WB-GC).

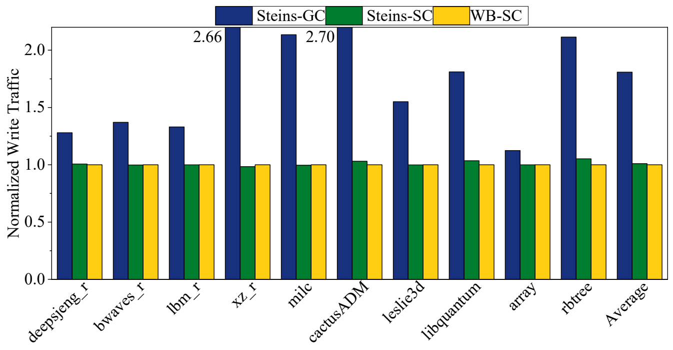

Fig. 14. The write traffic (normalized to WB-SC).

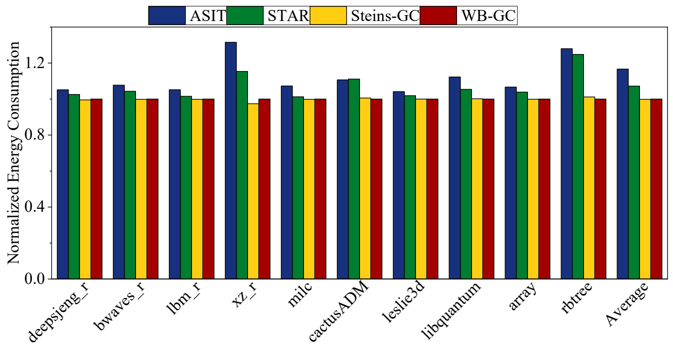

Fig. 15. The energy consumption (normalized to WB-GC).

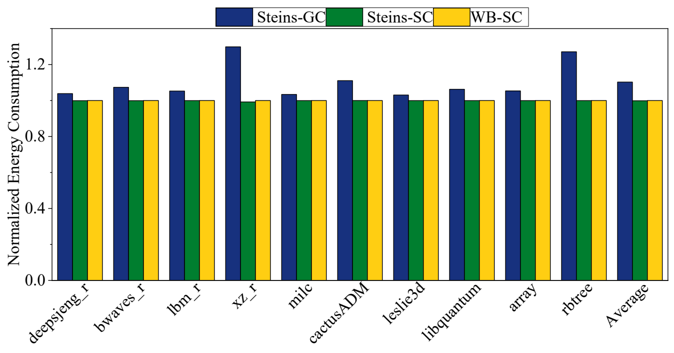

Fig. 16. The energy consumption (normalized to WB-SC).

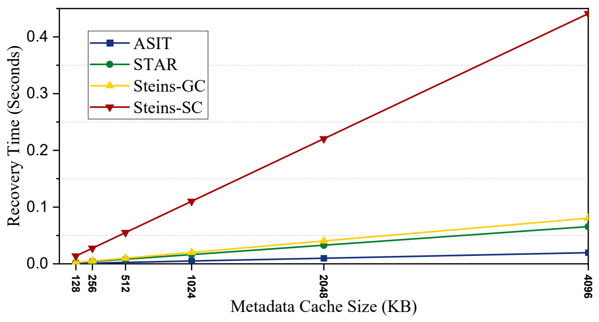

Fig. 17. The recovery time.

memory access, such as cactusADM, incur larger memory traffic than those with sequential access, such as lbm_r.

As shown in Fig. 14, compared to Steins-GC, the write traffic of Steins-SC is significantly decreased due to a higher cache hit ratio. Moreover, Steins-SC just incurs 1% extra write traffic compared to WB-SC.

C. The Energy Consumption

Steins eliminates the need to maintain a cache-tree, thereby reducing the number of HMAC calculations. Additionally, Steins has lower memory writes to NVM compared to ASIT and STAR as shown in IV-B. Fig. 15 illustrates the energy consumption across various schemes, highlighting that Steins-GC significantly reduces energy overhead compared to ASIT and STAR. Moreover, Steins-GC has lower energy consumption than WB-GC by 0.2% on average by optimizing metadata eviction. Additionally, Fig. 16 shows that Steins-SC can reduce the energy overhead by 9.4% compared to Steins-GC.

D. Recovery Time

To evaluate the recovery time, we assume that all of the cached metadata are dirty when the system crashes like existing works [16], [24], [49]. During recovery, Steins reads the offset records to locate the dirty nodes. As each record corresponds to a cache line in the metadata cache, the recovery time in Steins increases linearly with the metadata cache size. Given limited resources in the memory controller, we set a maximum cache size of 4MB to evaluate recovery time.

Similar to existing works [16], [21], [44], we assume that reading and verifying metadata from NVM consume 100ns.

To recover a dirty node, all the child nodes of this dirty node need to be fetched from NVM. For intermediate nodes with general counter block, Steins read 8 child nodes from NVM for recovery. To recover the split counter block in leaf nodes, Steins needs to read 64 data blocks. Moreover, Steins also needs to fetch the stale nodes stored in NVM and leverage their parent nodes for verification, which may incur iterative node reads. Since the calculation during recovery incurs low latency, the main recovery time is attributed to fetching the metadata from NVM.

Fig.17 shows the recovery time of different schemes. The baseline scheme, WB, does not support recovery. ASIT achieves the fastest recovery time, completing the recovery of 4MB security metadata in just 0.02s by persisting dirty metadata during runtime, at the cost of additional memory access. STAR requires 0.065s to recover 4MB of stale metadata by using the cache-tree for verification. Unlike them, our Steins delivers high performance and supports fast recovery. Specifically, the recovery time for a 4MB metadata cache is approximately 0.08s for Steins-GC and about 0.44s for Steins-GC. Considering that systems require between 10 to 100s to execute a self-test after system crashes [50], the recovery times for stale security metadata in Steins are deemed acceptable within existing high-availability systems [16], [21], [23].

E. Storage Overhead

To provide the integrity guarantee, all schemes need to store the entire SIT in NVM. With the general counter block, the leaf nodes of Steins-GC, ASIT, and STAR require 1/8 of the storage in 16GB NVM, i.e., 2GB. Steins-SC, which adopts the split counter block, requires only 1/64 of the space (256MB) for its leaf nodes. Additionally, because Steins-SC has fewer levels, the storage required for its intermediate nodes is also lower than that of the other schemes.

Moreover, STAR and ASIT require additional cache space for the cache-tree structure. In the 8-way metadata cache, STAR uses 8B HMAC for each set of the metadata cache. Therefore, STAR requires an extra 1/64 space to store the leaf nodes of cache-tree. ASIT introduces 8B HMAC for each 64B cache line and thus requires an extra 1/8 metadata cache space. Besides, both STAR and ASIT need a 64B on-chip non-volatile register to store the cache-tree root.

Unlike STAR and ASIT, Steins does not maintain a cache-tree for verification. Instead, Steins uses a 64B on-chip non-volatile register to store the $LInc$s and a 128B on-chip non-volatile buffer to temporarily store the generated parent counter.

F. Scalability of Steins

We implement Steins in memory controllers (MCs). The Optane DIMM connects to the processor's MC. For Intel's Cascade Lake processors, each processor has two MCs, each of which supports three Optane DIMMs [51]. When multiple clients access different DIMMs, their requests are executed in parallel in different MCs. If they initiate requests to the same DIMM, the requests are processed serially by our Steins.

V. Related Work

There are many schemes focusing on secure NVM systems. To improve performance, Synergy [52] replaces the ECC with the HMAC in ECC-DIMMs to persist the MAC and data in one memory access. Thoth [25] completes multiple partial security metadata updates in one memory access. To reduce the height and storage overhead of SIT, VAULT [22] and MorphCtr [42] compress the counter to store more counters in one counter block.

Prior schemes, such as SCA [45], Osiris [44], SecPM [53], and Supermem [40], study the consistency between the user data and counter block. SCA uses the ADR [54] to guarantee that the counter and user data can be flushed atomically. Hence, SCA does not need to recover the counters. Osiris employs a stop-loss mechanism to persist the counter. During the recovery process, Osiris attempts to recover the counter to the value within $[IV, IV+N]$ (IV is the stale counter stored in NVM) and leverages the ECC to verify the retrieved counter. Supermem [40] uses a write-through scheme to guarantee that counter and user data are flushed atomically. Steins can also leverage Osiris to recover the stale leaf nodes and then verify them using L_0Inc.

The BMF [55] and PLP [56] consider the consistency and recovery of BMT. PLP proposes the invariants that are required for recovery. Moreover, PLP leverages tree optimizations such as pipelined updates to reduce the overhead of atomically propagating modifications to the tree root. BMF uses an on-chip non-volatile metadata cache (nvMC) to split a large MT into multiple MTs and achieves better performance than PLP. ASIT, STAR, SCUE, and our Steins focus on the consistency and recovery of SIT. The evaluation results show that our Steins achieve higher performance and faster recovery while supporting lower storage overhead.

VI. Conclusion

In this paper, we propose Steins a novel approach designed to bridge the gap between fast recovery and high performance in secure NVM systems. Steins introduces an efficient counter generation scheme that enables the recovery of stale nodes from persistent child nodes. To accelerate the recovery process, Steins tracks the dirty nodes by their metadata region offsets and leverages the ADR to minimize NVM access. Moreover, after comprehensively analyzing the SIT with lazy update scheme, Steins maintains the $LInc$s for verification, which only incurs negligible overhead. Steins also enhances the write operations of dirty nodes, reducing data read/write latency. Compared with state-of-the-art schemes, Steins delivers higher performance and lower storage overhead while achieving fast recovery.

VII. Acknowledgments

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 62125202, U22B2022 and 61821003.

References

[1] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *IEEE micro*, vol. 30, no. 1, pp. 143–143, 2010.

[2] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: lightweight persistent memory," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 91–104, 2011.

[3] T. Ma, M. Zhang, K. Chen, Z. Song, Y. Wu, and X. Qian, "Asymnvm: An efficient framework for implementing persistent data structures on asymmetric nvm architecture," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 757–773.

[4] "Brief: Intel optane persistent memory." https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html.

[5] O. Patil, L. Ionkov, J. Lee, F. Mueller, and M. K. Lang, "Nvm-based energy and cost efficient hpc clusters," 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:238995682

[6] Y. Zhu, W. Yu, B. Jiao, K. Mohror, A. Moody, and F. Chowdhury, "Efficient user-level storage disaggregation for deep learning," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2019, pp. 1–12.

[7] W. Liu, H. Liu, X. Liao, H. Jin, and Y. Zhang, "Hngraph: Parallel graph processing in hybrid memory based numa systems," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2021, pp. 388–397.

[8] C. Foyer, B. Goglin, E. Jeannot, J. Klinkenberg, A. Kozhokanova, and C. Terboven, "H2m: Towards heuristics for heterogeneous memory," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2022, pp. 498–499.

[9] S. Chhabra and Y. Solihin, "i-nvmm: A secure non-volatile main memory system with incremental encryption," in *Proceedings of the 38th annual international symposium on Computer architecture*, 2011, pp. 177–188.

[10] 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*, 2019, pp. 104–115.

[11] Y. Xu, Y. Solihin, and X. Shen, "Merr: Improving security of persistent memory objects via efficient memory exposure reduction and randomization," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 987–1000.

[12] A. Awad, P. Manadhata, S. Haber, Y. Solihin, and W. Horne, "Silent shredder: Zero-cost shredding for secure non-volatile main memory controllers," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 263–276.

[13] V. Young, P. J. Nair, and M. K. Qureshi, "Deuce: Write-efficient encryption for non-volatile memories," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 33–44.

[14] J. Huang and Y. Hua, "Root crash consistency of sgx-style integrity trees in secure non-volatile memory systems," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 152–164.

[15] H. Lipmaa, P. Rogaway, and D. Wagner, "Ctr-mode encryption, comments to nist concerning aes modes of operations," in *NIST Workshop on Modes of Operation*, 2000.

[16] 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.

[17] J. Daemen and V. Rijmen, "Aes proposal: Rijndael," in *First Advanced Encryption Standard (AES) Conference*, 1998.

[18] 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.* IEEE, pp. 567–570.

[19] 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 *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2007, pp. 183–196.

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

[21] 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)*, 2021.

[22] 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*, 2018, pp. 665–678.

[23] 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.

[24] A. Freij, H. Zhou, and Y. Solihin, "Secpb: Architectures for secure non-volatile memory with battery-backed persist buffers," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 677–690.

[25] X. Han, J. Tuck, and A. Awad, "Thoth: Bridging the gap between persistently secure memories and memory interfaces of emerging nvms," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 94–107.

[26] Y. Zhang, X. Tu, L. Wang, Y. Hu, F. Wang, and Y. Wang, "Fullrepair: Towards optimal repair pipelining in erasure-coded clustered storage systems," in *2023 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2023, pp. 107–117.

[27] Y. Guo, W. Bland, P. Balaji, and X. Zhou, "Fault tolerant mapreduce-mpi for hpc clusters," in *proceedings of the international conference for high performance computing, networking, storage and analysis*, 2015, pp. 1–12.

[28] "The cost of downtime at the world's biggest online retailer," https://www.upguard.com/blog/the-cost-of-downtime-at-the-worlds-biggest-online-retailer, accessed November 20, 2023.

[29] "The cost of it downtime," https://www.the20.com/blog/the-cost-of-it-downtime/, accessed November 20, 2023.

[30] M. Jammal, A. Kanso, and A. Shami, "High availability-aware optimization digest for applications deployment in cloud," in *2015 IEEE International Conference on Communications (ICC)*. IEEE, 2015, pp. 6822–6828.

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

[32] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[33] 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.

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

[35] "Spec releases major new cpu benchmark suite," https://www.spec.org/cpu2017/press/release.html, accessed November 20, 2023.

[36] P. Saravanan and P. Kalpana, "An energy efficient xor gate implementation resistant to power analysis attacks," *J. Eng. Sci. Technol*, vol. 10, no. 1, pp. 1275–1292, 2015.

[37] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom *et al.*, "Meltdown: Reading kernel memory from user space," *Communications of the ACM*, vol. 63, no. 6, pp. 46–56, 2020.

[38] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020.

[39] 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.

[40] P. Zuo, Y. Hua, and Y. Xie, "Supermem: Enabling application-transparent secure persistent memory with low overheads," in *Pro-*

ceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019, pp. 479–492.

[41] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 179–190, 2006.

[42] 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.

[43] H. Li, R. Lu, L. Zhou, B. Yang, and X. Shen, "An efficient merkle-tree-based authentication scheme for smart grid," *IEEE Systems Journal*, vol. 8, no. 2, pp. 655–663, 2013.

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

[45] 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.

[46] F. Huang, D. Feng, Y. Hua, and W. Zhou, "A wear-leveling-aware counter mode for data encryption in non-volatile memories," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, pp. 910–913.

[47] 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.

[48] 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.

[49] X. Han, J. Tuck, and A. Awad, "Horus: Persistent security for extended persistence-domain memory systems," in *55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 1255–1269.

[50] "Power-on self-test," https://en.wikipedia.org/wiki/Power-on self-test.

[51] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020, pp. 169–182.

[52] 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.

[53] 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.

[54] A. Rudoff, "Persistent memory programming," *Login: The Usenix Magazine*, vol. 42, no. 2, pp. 34–40, 2017.

[55] A. Freij, H. Zhou, and Y. Solihin, "Bonsai merkle forests: Efficiently achieving crash consistency in secure persistent memory," in *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021, pp. 1227—1240.

[56] A. Freij, S. Yuan, H. Zhou, and Y. Solihin, "Persist-level parallelism: Streamlining integrity tree updates for secure non-volatile memory," in *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 14–27.