A Write-Friendly and Fast-Recovery Scheme for Security Metadata in Non-Volatile Memories

Jianming Huang and Yu Hua*
Wuhan National Laboratory for Optoelectronics
School of Computer, Huazhong University of Science and Technology
Email: {jmhuang, csyhua}@hust.edu.cn

***Abstract*—Non-Volatile Memories (NVMs) require security mechanisms, e.g., counter mode encryption and integrity tree verification, which are important to protect systems in terms of encryption and data integrity. These security mechanisms heavily rely on extra security metadata that need to be efficiently and accurately recovered after system crashes or power off. Established SGX integrity tree (SIT) becomes efficient to protect system integrity and however fails to be restored from leaves, since the computations of SIT nodes need their parent nodes as inputs. To recover the security metadata with low write overhead and short recovery time, we propose an efficient and instantaneous persistence scheme, called STAR, which instantly persists the modifications of security metadata without extra memory writes. STAR is motivated by our observation that the parent nodes in cache are modified due to persisting their child nodes. STAR stores the modifications of parent nodes in their child nodes and persists them just using one atomic memory write. To eliminate the overhead of persisting the modifications, STAR coalesces the modifications and MACs in the evicted metadata. For fast recovery and verification of the metadata, STAR uses bitmap lines in asynchronous DRAM refresh (ADR) to indicate the locations of stale metadata, and constructs a cached merkle tree to verify the correctness of the recovery process. Our evaluation results show that compared with state-of-the-art work, our proposed STAR delivers high performance, low write traffic, low energy consumption and short recovery time.**

I. Introduction

Non-Volatile Memory (NVM) is becoming the main devices of next-generation memory systems, due to high density, near-zero standby power, non-volatile and byte-addressable features. NVM also suffers from limited cell endurance [19], [26], [39], [43] and asymmetric read and write performance [16], [31], [42]. The write energy consumption of NVM is 2 times that of DRAM [14]. More importantly, due to non-volatile property, NVM has to handle severe security vulnerabilities. After physically stealing DIMM, an attacker can easily read the contents via another computer due to retaining data after power off in NVM. To protect the user data from attacks, an intuitive solution is to leverage data encryption and integrity verification schemes, which unfortunately introduce extra security metadata, respectively including counter blocks and integrity tree nodes. To support the normal execution of applications after crashes, it is important to recover these metadata to a consistent state [7], [38], [40], [44].

In general, counter mode encryption and integrity tree are used to protect NVM systems respectively for security and integrity verification. Due to hiding the decryption latency, counter mode encryption (CME) [22] for secure memory systems becomes more efficient via one-time padding than direct encryption via AES [6], [33], [41], [46]. To protect the integrity of data, an integrity tree is used in memory systems [7], [28], [44]. The counter blocks in CME are hashed to generate the Message Authentication Codes (MACs) that are further hashed iteratively until generating a root node that is stored in the on-chip non-volatile register, called Bonsai Merkle Tree (BMT) [28]. Unlike BMT, SGX integrity tree (SIT) [12], [34] consists of eight counters and one MAC in each node. The MAC of one node is generated by hashing the eight counters in the node, the address of the node and one corresponding counter in the parent node. The security metadata, generally including counter blocks and integrity tree nodes, need to be recoverable to ensure the system security and integrity after system crashes. In the BMT and SIT, the counter blocks are the leaf nodes of the integrity tree. Thus every metadata except the root has its parent node in the integrity tree, and the counter blocks are the parent nodes of their encryption user data. SIT updates MACs in each node in parallel, while BMT sequentially updates MACs in the same branch. Moreover, due to containing counters in nodes, the nodes in SIT can be compacted to reduce the memory bandwidth consumption [29], [34]. For better performance, we leverage the SIT to protect NVM systems. In this paper, each security metadata, i.e., the counter block and integrity tree node, is 64 bytes, that match the cache line granularity.

To improve the system performance, we need to create security metadata cache in the memory controller to cache the metadata. If dirty metadata in the cache are not persisted before system crashes, the metadata become stale in NVM. And these stale metadata can't be used to ensure the security of systems after reboot. The encrypted systems only need to persist the counter blocks [23], [40], [45]. However, in the security systems with integrity tree, multi-level tree nodes also need to be persisted. In the integrity tree, persisting modified tree nodes causes the modifications of the ancestor nodes in the metadata cache, which leads to new inconsistencies between cache and NVM. After system crashes, the metadata need to be recovered into the latest consistent state. Note that the user data also need to be recovered into a consistent state, which can be addressed via log and PMDK [3].

The persistent memory requires short recovery time and low

*Corresponding author.

2378-203X/21/$31.00 ©2021 IEEE
DOI 10.1109/HPCA51647.2021.00038

write overhead. In Amazon's cloud system, the downtime costs are up to 2 million dollars per minute [1], and the average costs of IT downtime are 5,600 dollars per minute [2]. Moreover, for high-availability requirement systems that need to meet the availability target of 99.999% (five nines rule), such as bank systems and online transactions, short recovery time is necessary. As shown in [44], recovering metadata needs several hours without fast recovery schemes. Long recovery time makes the normal recovery scheme inefficient [7]. On the other hand, the high write latency, high energy consumption and limited write endurance of NVM require less write overhead during the running time.

Prior works can't both achieve the short recovery time and low write overhead. Osiris [40], Supermem [45] and SCA [23] have low write overhead. However, they only focus on counter blocks and can't be used in the integrity tree. Triad-NVM [7] extends the recovery ability to the integrity tree by persisting the low-level tree nodes and reconstructing the tree from the persisted low-level nodes. However, Triad-NVM fails to recover the SGX integrity tree (SIT) since the computation of SIT nodes' MACs needs the counters in parent nodes as inputs. On the other hand, Triad-NVM needs 2–4 times memory writes to persist the low-level tree nodes. Anubis [44] has the ability to recover BMT and SIT. However, for recovering SIT, Anubis needs 2 times memory writes to persist the modifications of metadata, which incurs the extra write traffic, performance overhead and energy consumption. Moreover, strict persistence schemes [7], [44] for persisting all modified metadata via write-through need no recovery after crashes, since there are no stale metadata. Unfortunately, the strict persistence schemes incur tens of times the write amplification, which is unacceptable in NVM due to limited write endurance and high write energy consumption.

To achieve short recovery time after system crashes and low write overhead on running time, we propose STAR (**S**IT **t**race **a**nd **r**ecovery scheme) to instantly persist modifications of metadata and fast recover the stale metadata on recovery. The insight behind STAR is that the modifications in metadata cache are caused by persisting one metadata. To avoid two writes respectively for the metadata and modification, we coalesce them into one write maintained in the metadata to be persisted. For example, persisting metadata A causes the modifications of its parent node. We store these modifications in the metadata A, so that we atomically persist A and the modifications in one write, without extra memory writes.

We observe that ① in SIT, the modifications caused by persisting metadata occur on the parent node of the persisted one; and ② the 64-bit MAC in data contains 10-bit unused space. Based on the two observations, STAR coalesces the modifications and persisted metadata in one memory write (Section III-B). And the modifications of metadata are instantly persisted. For fast recovery, it is necessary to distinguish the stale metadata that need to be recovered from the metadata space. To identify the stale metadata after crashes, STAR places 16 bitmap lines within asynchronous DRAM refresh (ADR) in the memory controller to indicate which metadata are stale in NVM (Section III-C). The bitmap lines stored in ADR can be flushed into the recovery area in NVM by battery backup support when systems power off. To further accelerate the recovery process, STAR introduces a multi-layer index structure to selectively read the useful bitmap lines from the recovery area (Section III-D). Moreover, STAR constructs a cache-tree and uses its root to verify the correctness of the recovery process (Section III-E).

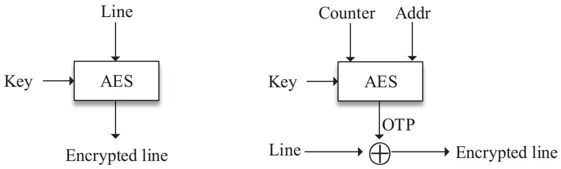
To evaluate the performance of our proposed scheme, we use Gem5 [8] with NVMain [25] to implement STAR. To show the effectiveness of our STAR, we evaluate 5 persistent micro-benchmarks that have been widely used in existing works [7], [11], [17], [18], [23], [27], [45] and 2 macro-benchmarks from WHISPER [24]. Our experimental results show that STAR reduces 92% extra memory write traffic compared with state-of-the-art work, Anubis. STAR also reduces the IPC overhead from 10% to 2% and significantly reduces the energy consumption from 46% to 4%. STAR needs about 0.05s to recover the security metadata in the systems with a 4MB metadata cache. Since systems generally require 10–100s to execute self-test [4] after system crashes, our metadata recovery time is negligible in the real-world recovery process. Moreover, the recovery time in STAR is proportional to the number of dirty metadata in metadata cache, instead of NVM or cache sizes, which offers the adaptability for the larger NVM and metadata cache sizes. In summary, this paper makes the following contributions:

- **Counter-MAC synergization for persisting modifications in security metadata cache.** We explore and exploit the unused bits in the MAC of data to store the modifications of its parent node, which offers a new approach to consistently persisting the modifications in metadata cache and restoring the stale metadata without extra memory writes.
- **Bitmap lines for locating stale metadata.** To efficiently locate the stale metadata with low write overhead, we propose bitmap lines to absorb the location information. Using the bitmap lines to record location is useful for applications with high spatial localities.
- **Cache-tree for verifying the process of recovery.** We analyze the challenges of constructing a merkle tree on cache, which ensures the correctness of recovery. To address the challenges, we introduce the cache-tree based on the set-way structure of cache to detect the attacks on restored metadata with low computation overhead.
- **Extensive experiments.** We have implemented and evaluated STAR via persistent micro- and macro-benchmarks, and experimental results show STAR reduces the extra memory write traffic by up to 92% with low recovery time compared with state-of-the-art work. STAR also improves system performance and reduces energy consumption.

II. Background and Motivation

A. Threat Model

In general, only the processor chip is considered safe in our threat model similar to the state-of-the-art works [6], [7], [23], [28], [36], [40], [44]. An attacker attacks the memory



(a) Straightforward encryption (b) Counter mode encryption

Fig. 1. The comparisons of encryption schemes.

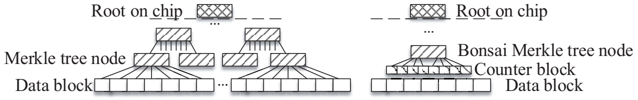
via multiple methods, such as scanning the memory, snooping the memory bus and stealing DIMM to obtain the user data, which exacerbate the data confidentiality. Moreover, attackers also replay memory data and tamper with memory contents, which undermine the data integrity. In this paper, we use counter mode encryption (CME) and SGX integrity tree (SIT) to defend against these attacks. Other attacks such as access pattern leakage, power analysis and side-channel attacks, are beyond the scope of this paper.

B. Counter Mode Encryption

Since NVMs retain data after power failures, information leakage in NVMs is more severe than the volatile DRAM. Data encryption becomes necessary to guarantee data confidentiality, which can be processed in memory [9] or processor side [36]. However, in the memory-side encryption, the plaintext data have to pass through the memory bus and are snooped by attackers. Thus, prior works use the encryption in the processor side [6], [33], [41], [46]. A straightforward method in the processor side to encrypt a memory line is to use a block cipher algorithm, e.g., AES, with a global key, as shown in Fig. 1(a). However, this scheme suffers from some limitations. Due to the unchanged keys, attackers can easily break the encryption by using dictionary attacks. Moreover, the decryption process exists on the read critical path. The ciphertext data read from NVM have to be first decrypted, which causes a long decryption latency.

Counter mode encryption (CME) is proposed to compensate for the above drawbacks. As shown in Fig. 1(b), CME first uses a counter, a data line address and a global secret key to generate a one-time padding (OTP) via the AES algorithm. For memory writes, the cache line to be written needs to be encrypted by XORing the plaintext data and OTP. For memory reads, OTP is generated via the cached counter block in parallel with reading a memory line, and the plaintext data are obtained by XORing the memory line and OTP. Thus the decryption latency is hidden by the latency of reading data.

To provide high security, OTP will not be reused. To meet this requirement, OTP generation uses three inputs, i.e., the line address, counter and key. Different data lines have different addresses, which allows the OTPs of different lines not to be reused. For the same line, each memory write causes its counter to increase by one, and thus OTP will not be reused in the same line at different writes. In general, each counter block contains 64 7-bit minor counters and one 64-bit major counter. Counter blocks are cached in the memory controller and persisted in NVM. One counter block covers 64 user data blocks, i.e., one page. CME encrypts data blocks using the corresponding minor counter and major counter. When one minor counter overflows, the major counter increases by one. All the minor counters are reset and the data blocks in this page need to be re-encrypted. The 64-bit major counter never overflows throughout the lifespan of an NVM, since the count range, i.e., $2^{64} \approx 10^{20}$, is much larger than the endurance limit of NVM cell, e.g., 10^7-10^9 for PCM [26], [43] and 10^8-10^{12} for ReRAM [20], [21].



(a) Merkle tree (b) Bonsai Merkle tree

Fig. 2. Different bottom-up integrity trees. (a) Merkle tree starts hashing from data blocks; (b) Bonsai Merkle tree starts hashing from counter blocks.

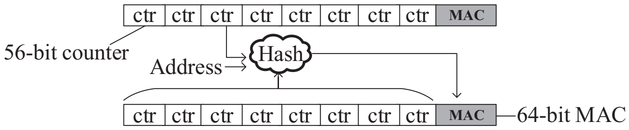


Fig. 3. SGX integrity tree (SIT) generates the MAC by hashing node address, all counters in this node and one corresponding counter in the parent node.

C. Integrity Tree

Attackers can replay the data, counter and MAC [28]. To protect data integrity, integrity trees have been widely used in memory systems, e.g., merkle tree, bonsai merkle tree (BMT) and SGX integrity tree (SIT) [12], [15], [28], [34]. In the merkle tree, as shown in Fig. 2(a), several data blocks are hashed together to generate MAC using a cryptographic hash function stored in the processor. Higher level MAC is generated by hashing the lower level MACs together and finally, the merkle tree is formed with one 64B hash root stored on chip. Bonsai merkle tree generates the first MAC level by hashing counter blocks instead of user data as shown in Fig. 2(b). Since the number of counter blocks is much smaller than that of user data blocks, bonsai merkle tree has fewer leaf nodes and lower heights than merkle trees. Since each MAC in one node leverages the MACs of its child nodes as inputs, the (bonsai) merkle trees calculate their MACs sequentially, i.e., the tree can't calculate the MAC of one node before completing the computation upon the MACs of its child nodes.

Different from merkle tree and bonsai merkle tree, an SIT node contains eight 56-bit counters and one 64-bit MAC [34], instead of hash values. Like counter blocks, an SIT node is 64-byte and stored in memory/cache line granularity. As shown in Fig. 3, the MAC in each SIT node is generated by hashing the node address, the counters in this node and one corresponding counter in the parent node. SIT can calculate different level MACs in parallel as long as these counters have been prepared. In the SIT-based systems, the counter blocks in CME are the leaves of SIT, and the counter blocks have the same structures as SIT nodes, i.e., eight counters and one MAC.

In SIT, when one SIT node is read into cache, systems recalculate the MAC and compare it with the MAC stored in the SIT node to detect whether the node is attacked. SIT has two options to update the tree nodes, including eager and lazy schemes. The eager scheme is to propagate the changes to the root and modify root immediately when one data block is flushed into NVM. Persisting data blocks causes the corresponding counters in all ancestor nodes to increase by one. The root exhibits the data changes and has a higher probability of overflowing, since each user data write, no matter where it is written, causes a counter in the root to increase. Specifically, in the lazy scheme, once a data block, i.e., the user data block, counter block or SIT node, is flushed into NVM, its ancestor SIT nodes are cached to verify the integrity of the flushed data block. The corresponding counter in the parent node further increases by one, and the MACs in the flushed data block and parent node are hence modified. The other ancestor nodes, including the root, are unchanged. These nodes are modified only when their dirty child nodes are flushed into NVM from the cache. The SIT root is not modified immediately with the changes of data and intermediate nodes in the lazy scheme. The cached tree nodes are treated as trusted bases since they have been verified when they enter the cache. After crashes, since the cached nodes are lost and root is lazy updated, the attacks occur during recovery can't be detected by SIT. Thus the systems need a new approach to verifying the correctness of recovery [44].

Due to fewer computations of MACs than the eager scheme, we leverage the lazy scheme to update the SIT, like Synergy [30], Vault [34] and Anubis [44].

D. The MACs in Persistent Memory

SIT uses MACs stored in tree nodes (including counter blocks, which are the leaves of SIT) to associate one tree node with its parent node. Any unauthorized modifications over counters and MACs in nodes could be detected by comparing the stored and calculated MACs. The user data also need MACs to associate the data with encryption counters by hashing the data, data address and corresponding counter in counter block to generate the MAC. Without the MAC, an illegal modification in the user data can't be detected, and the wrong data will be used by CPU after decryption. When reading user data, the integrity of the user data needs to be verified by using MAC. To avoid the failure of integrity checking after recovery, MACs need to be written into NVM with the new user data. However, the MAC of user data is logically placed with the user data line, but physically placed in another memory line. To persist MAC and user data atomically and decrease the number of the accesses to MAC memory lines, Synergy [30] stores the MAC in the 9th chip, in which the Error Correction Code (ECC) is previously stored. Synergy reads/writes the data and MAC in one memory access. Similar to Synergy, we store the user data and MAC in one line, instead of two memory lines.

In general, the size of MAC is 64 bits [34]. However, 54-bit MAC is also safe as described in Morphable Counters [29]. There are 10 unused bits in a 64-bit MAC space. We will reuse these unused bits in our design to instantly persist the modifications of security metadata.

E. Motivation

The state-of-the-art works provide the metadata recovery in the persistent memory after system crashes, e.g., Osiris [40] Triad-NVM [7] and Anubis [44]. However, Osiris and Triad-NVM fail to support SIT recovery, and Anubis causes high write overhead. We need to fast recover the security metadata with low write overhead in the SIT-based persistent systems.

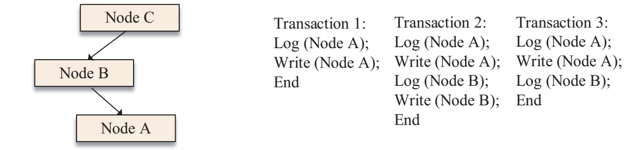
Osiris relaxes the counter block persistence during system running time. After system crashes, Osiris recovers the stale counter block by checking the counter from the *stale counter* to *stale counter+N* to find the correct counter. Osiris checks the correctness of each alternative counter by computing ECC and finally detects the data replay attack by using the merkle tree root stored on chip. However, Osiris doesn't discuss how to persist and recover the SIT nodes. Moreover, in the widely used SIT lazy scheme, the root doesn't demonstrate the latest data in memory after crashes. Attackers can simply replay the data, MAC and ECC with old tuple on recovery, and this data replacement can't be detected when Osiris is used for SIT. Osiris also needs a long time to recover all counter blocks due to failing to distinguish between the stale and fresh counter blocks.

Triad-NVM forces to persist multi-level merkle tree nodes with the user data into NVM. The merkle tree recovery on Triad-NVM needs to reconstruct the whole merkle tree from the leaves and compare the reconstructed root with the one stored on chip. However, SIT can't be constructed from the leaves. Without the correct corresponding counter in the parent node, the MAC in one SIT node can't even be computed. Due to forced persistence of the low-levels merkle tree nodes, Triad-NVM incurs 2–4 times write overheads.

Anubis provides fast recovery schemes for both merkle tree and SGX integrity tree. For a merkle tree, when a metadata block in cache is marked dirty from a clean state, an extra shadow table (ST) block with this dirty block address is written into NVM. For the SGX integrity tree, each metadata write from the cache into memory incurs an extra ST block write containing the address, the LSBs of counters and MAC of the parent node of the written metadata. To ensure the consistency of ST and dirty security metadata blocks, Anubis needs an atomic scheme to persist the ST and security metadata blocks like SCA [23] and Supermem [45]. The Anubis recovery scheme for SIT has 2 times memory writes compared with a normal write-back scheme, thus increasing the write traffic and the energy consumption of NVM.

Concurrent work, Phoenix [5], contains the updates of counter blocks in the cache tree root and doesn't persist the modifications of counter blocks, which can be recovered via Osiris [40]. Unlike Phoenix, our STAR removes the extra writes of the whole tree, including the counter blocks and intermediate tree nodes, by containing the changes in the unused bits of the MAC fields.

In addition to the above works, logs are usually used to restore the inconsistent data in systems, e.g., redo log and undo log. For the redo log, the new data are first written into the log, and then the old data are updated in-place. If system crashes occur during writing logs, the old data in-place are consistent. If the crashes occur during updating old data, the inconsistent old data can be recovered according to the new



Transaction 1:	Transaction 2:	Transaction 3:
Log (Node A);	Log (Node A);	Log (Node A);
Write (Node A);	Write (Node A);	Write (Node A);
End	Log (Node B);	Log (Node B);
	Write (Node B);	End
	End	

(a) Partial SIT in metadata cache (b) Three log schemes

Fig. 4. Using log schemes to persist the SIT nodes.

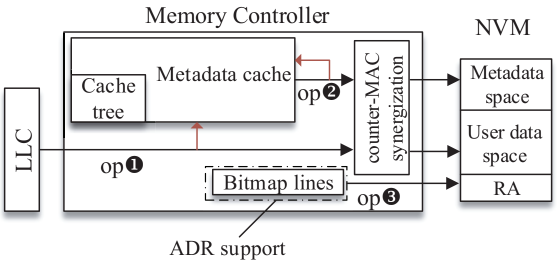


Fig. 5. The overview of STAR design. (op ❶) Flushing user data with counter-MAC synergization incurs the modifications in metadata cache. (op ❷) Flushing metadata with counter-MAC synergization incurs the modifications in metadata cache. (op ❸) Bitmap lines are flushed into recovery area (RA) by LRU policy. Cache-tree exhibits changes in metadata cache.

data recorded in the log. An undo log is also used to recover the data by the old data in a log. However, SIT fails to use logs to ensure the consistency between two tree nodes. As shown in Fig. 4(a), node B is the parent node of node A, and node C is the parent node of node B. The metadata cache needs to evict the node A due to the cache replacement policy. Fig. 4(b) shows three different log schemes for persisting node A. Transaction 1 ensures that the node A itself in NVM is consistent. However, the node B has been modified due to the eviction of its child node A. After system crashes, node B in NVM is stale and needs to be restored. Transaction 2 provides the consistencies of both nodes A and B, but the node C is stale after system crashes. Transaction 3 writes the node A and logs the node B. When crashes occur, although the node B is stale, it could be restored from the log. This transaction provides the consistency of nodes A and B. However, this log scheme is similar to Anubis with extra 1x writes, i.e., when writing node A, Anubis uses the ST node to record and restore the modified node B, while Transaction 3 uses the log.

Existing works provide different approaches to recovering the security metadata. However, some can't be used in SIT, and others incur high write and recovery overheads. Unlike existing works, we focus on SIT lazy update scheme and counter mode encryption to propose a new scheme STAR, which reduces the extra memory writes on running time with fast recovery after system crashes while offering crash consistency between metadata and metadata/user data.

III. System Design and Implementations

A. *STAR Overview*

To fast recover security metadata after system crashes with low write overhead, our paper proposes STAR that instantly persists the modifications in metadata cache and restores the stale metadata from their child nodes after crashes. Persisting the child node into NVM only modifies the parent node. STAR hence stores the modifications of the parent node in the unused bits of the child node's MAC. The modifications of the parent node are instantly persisted after modifying the parent node, without any extra memory writes. After system crashes, STAR obtains the modifications of the stale parent node from its persistent child node. The parent node is restored by combining its stale version and the modifications.

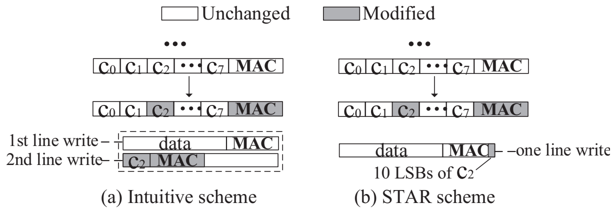
In the context of SIT, our STAR recovers the stale security metadata in a short time with low write overhead. As shown in Fig. 5, STAR consists of the counter-MAC synergization, bitmap lines and cache-tree structure. The user data flushing from LLC into NVM causes the modifications of the counter blocks, i.e., the parent nodes of user data (op ❶). The counter blocks/SIT nodes flushing from metadata cache in memory controller into the metadata space also causes the modifications of their parent nodes (op ❷). When flushing metadata and user data, STAR stores the modifications in the child node's MAC space and the modifications are persisted with child node, called counter-MAC synergization. The counter-MAC synergization leverages the unused bits in the MAC space of one node to store the modifications of its parent node. During recovery, the stale metadata are restored from their child nodes.

To fast recover the metadata, STAR uses bitmap lines to record the locations of stale metadata. On recovery, STAR restores the stale metadata instead of all metadata, thus consuming short recovery time. Bitmap lines are stored in an asynchronous DRAM refresh (ADR) region in the memory controller and flushed into the Recovery Area (RA) in NVM by the least recently used (LRU) policy (op ❸).

Finally, to verify the correctness of the recovery process, i.e., whether an attack occurs, the cache-tree in the metadata cache is constructed. The dirty metadata in cache will be stale when crashes occur, since they are not persisted into NVM in time. The cache-tree is constructed via the dirty metadata in the cache. The root of the cache tree is stored in the non-volatile register on chip. If an attack occurs during the recovery process, the stale node (i.e., the dirty node in cache) can't be restored to the latest state. The root of the reconstructed cache-tree will not match the stored root, hence identifying the occurrence of attacks.

B. *Persisting the Modifications of Metadata*

The security metadata in the cache are divided into two categories, i.e., clean and dirty. Specifically, the clean metadata in cache are the same as their counterparts in NVM. The dirty metadata in cache are different from their counterparts in NVM, since the cached metadata have been modified but not flushed into NVM. If dirty metadata in the cache are not persisted before system crashes, the corresponding counterparts become stale in NVM. After system crashes, we only need to restore the stale metadata in NVM. A cache has a 'dirty bit' for each cache line to indicate whether this cache line is dirty. As described in Section I, each metadata is 64B



(a) Intuitive scheme (b) STAR scheme

Fig. 6. Persisting the modifications of dirty nodes. (a) The intuitive scheme persists two lines with atomicity assurance; (b) STAR scheme persists one line via counter-MAC synergization.

and stored in a cache line. We distinguish the clean and dirty metadata in cache by checking the dirty bits of the cache lines.

After system crashes, we need to restore the stale metadata. One key observation is that in SIT lazy update scheme (detailed in Section II-C), when one data block is evicted from cache, only the corresponding counter in its parent node increases by one, and the MAC in the parent node is hence modified. Other counters in the parent node and other nodes are unchanged as shown in Fig. 6. The systems need to persist the modifications. On recovery, after obtaining the modifications of the stale node's counter, we restore the node by combining the modifications and the stale version in NVM. Finally, the MAC of the stale node is recomputed.

To persist modifications of the parent node in time, one intuitive scheme is to store the modified counter and MAC in one line, and persist the line with data block in an atomic operation, as shown in Fig. 6(a). However, the intuitive scheme incurs 2x memory writes and requires atomicity assurance like SCA [23].

Unlike the intuitive scheme, we store the modifications of metadata in their child nodes' MAC space, as shown in Fig. 6(b). As described in Section II-D, 54-bit MAC is safe [29], and 10 bits are unused in the 64-bit MAC space. STAR leverages these unused bits of MAC in the child node to store 10 LSBs of the corresponding counter in the parent node.

When one user data to be written arrives at the memory controller, only the corresponding counter in the counter block increases by one. STAR stores the 10 LSBs of the increased counter in the unused space of user data's MAC, called *counter-MAC synergization*. The counter block contains eight counters and has eight child user data. The LSBs of counter are stored in the corresponding user data's MAC. Moreover, the MAC and user data are organized in one line [30]. Hence the modifications of the counter blocks are atomically persisted with the user data without any atomicity assurance.

The counter-MAC synergization is also leveraged in security metadata. When one dirty metadata, i.e., counter block or SIT node, is evicted from metadata cache in the memory controller, the corresponding counter in the parent node increases by one. The LSBs of the increased counter are stored in the unused space of the evicted metadata's MAC space and persisted with the evicted metadata. Thus the modifications of the security metadata are instantly persisted once be generated.

To protect the LSBs, MAC in a data block is computed by hashing the content in the block, the address of the block, the corresponding counter in the parent node and the LSBs stored in the MAC space. When a counter in one metadata has been increased 2^{10} times, the metadata needs to be flushed into NVM to update the Most Significant Bits (MSBs), which are used on recovery to restore the stale metadata. This counter overflow is rare and introduces negligible overhead.

After system crashes, the metadata are restored in a bottom-up manner. The counter blocks are restored from the corresponding user data which are the child nodes of counter blocks, and the SIT nodes are restored from their child nodes. For one stale metadata, STAR obtains the correct LSBs of counters from its child nodes' MAC space. Even if the parent and child nodes are both dirty in cache, the recovery is also correct. The parent node becomes dirty due to persisting the child node. Thus the modifications of the dirty parent node are flushed with the child node via counter-MAC synergization. If the child node is dirty in cache, since the dirty node has not been evicted, the corresponding counter in parent node is not modified. We don't need to recover the counter after crashes.

Combined the MSBs stored in NVM with the LSBs obtained from the child node's MAC, counters in the stale metadata are restored. According to the corresponding counter in the parent node (if necessary, the parent node also needs to be restored), the MAC in this stale metadata is recomputed. With the counters and MACs, the stale metadata are recovered.

However, restoring all metadata requires a long recovery time. For fast recovery, STAR uses bitmap lines to track the locations of the stale metadata in Section III-C. Moreover, restoring nodes according to counter-MAC synergization suffers from data replay attacks. For example, when recovering counter blocks from their child user data, attackers replace the user data, MACs and LSBs in child data with an old tuple. These attacks can't be detected since the stale counter block matches the replayed child data. STAR needs to verify the correctness of the recovery process. We describe the verification mechanism in Section III-E.

C. Tracking the Locations of Stale Metadata

The stale metadata need to be clearly identified via their locations to facilitate fast recovery. Without the locations, all metadata in NVM have to be fully restored even if some are not stale, which requires a long recovery time. To efficiently record the locations of the stale metadata, we use the bitmap lines within ADR in memory controller. One bit in the bitmap line represents a security metadata line, e.g., the first and last bits in the first bitmap line represent the 1st and 512th metadata lines in metadata space. Each bitmap line covers 32KB continuous metadata space since one line contains 512 bits (512×64B=32KB).

The bitmap lines are used for recovery to indicate the locations of the stale metadata. During system running, when the cached metadata lines become dirty, the corresponding bits in bitmap line are set to 1, i.e., the dirty metadata become stale in NVM due to not being flushed into NVM. When the cached dirty metadata lines are flushed into NVM, STAR resets the corresponding bits to 0, i.e., the metadata in NVM are not

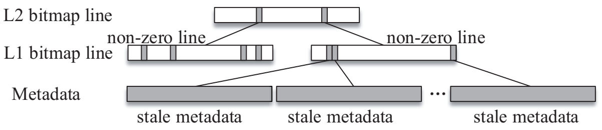


Fig. 7. A multi-layer index structure is used to read the non-zero bitmap lines and stale metadata.

stale. If the cached dirty metadata lines are modified again, STAR doesn't access the bitmap lines to change the bits, since the corresponding bits have been set to 1 when the metadata lines become dirty. Moreover, when the cached clean metadata lines are evicted from the cache, STAR doesn't access the bitmap lines, since the corresponding bits are already 0. In summary, we only access the bitmap lines when the dirty states of metadata lines change (from clean to dirty or from dirty to clean). According to the dirty bit in the cache, it is easy to determine whether the dirty states change.

The bitmap lines need to be persisted when system crashes occur. To ensure the persistence of bitmap lines, STAR leverages the ADR mechanism. Modern processor vendors provide a small battery backup for ADR with tens of entries [23], [32] in the memory controller. When the system crashes occur, the data stored in ADR are flushed into NVM by battery backup support. STAR places a certain number of the bitmap lines in ADR (the default value is 16 lines) so that these bitmap lines would be persisted when crashes occur. Extending ADR to store metadata is also used in SCA [23], which doesn't impact the performance but consumes the on-chip space.

If the bitmap lines in ADR don't cover the metadata line whose dirty state changes, i.e., the corresponding bit exists in another bitmap line in NVM, STAR flushes one bitmap line from ADR to the Recovery Area (RA) in NVM by LRU policy and reads the bitmap line into memory controller from RA to record the location of state-changed metadata line. For example, when a metadata line becomes dirty from clean, STAR needs to access the corresponding bitmap line. If there is no corresponding bitmap line in ADR, STAR reads the corresponding bitmap line from RA into memory controller and flushes one bitmap line into RA via LRU. As shown in Fig. 10 in Section IV, the frequency of writing/reading bitmap lines is low. Thus the overhead is negligible. The RA in NVM is allocated to store all bitmap lines, and consumes another negligible 1/512 metadata space.

D. Using Multi-layer Index to Speed Up Recovery

To locate the stale metadata in NVM during recovery, STAR needs to read all bitmap lines in RA. For a 16-GB NVM, the size of RA is 1/512 of the size of metadata space in NVM, i.e., RA occupies a 4-MB NVM space. However, reading the 4-MB RA also causes long latency. To speed up the STAR recovery process, we observe that only the locations of the stale metadata are necessary. Even if all metadata in the metadata cache are dirty and not written into NVM when crashes occur, many bitmap lines in RA are useless for recovery. The bitmap lines are designed to cover all metadata space, which is much larger than the metadata cache. Since the zero lines don't record the locations of stale metadata, reading these lines from RA is useless and time-consuming during recovery. It is more efficient to only read the non-zero bitmap lines, instead of reading all lines from RA.

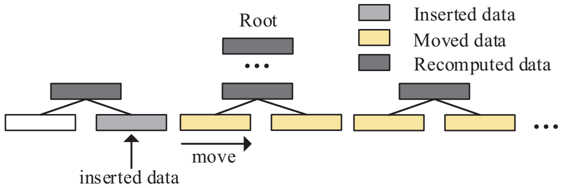


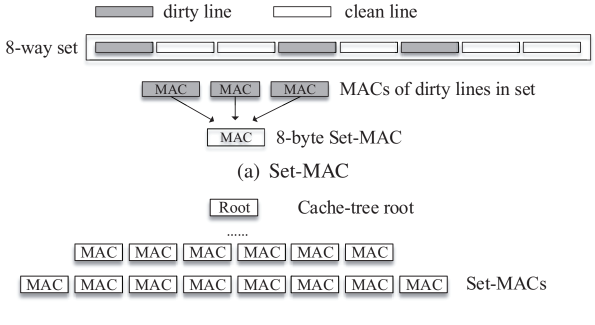
Fig. 8. Inserting a leaf node causes the changes of the whole merkle tree.

To speed up reading RA, we propose a multi-layer index, as shown in Fig. 7. STAR leverages L1 bitmap lines to indicate which security metadata lines are stale and L2 bitmap lines to indicate which L1 bitmap lines are non-zero. If necessary, STAR can add L3 bitmap lines and so on. We call this structure a *multi-layer index*. STAR stores the highest-layer bitmap lines in the on-chip non-volatile register like SIT root, and never flushes these lines into NVM. To reduce the consumption of on-chip space, the number of bitmap lines in the highest layer is always one. Other-layer lines exist in ADR in the memory controller and are flushed into RA by LRU, like the bitmap lines described in Section III-C, with lightweight NVM space consumption. A 1-/2-/3-layer index can cover 32KB/16MB/8GB metadata space. In our evaluation, we model 16GB main memory (about 2GB metadata), and the 3-layer index is sufficient to cover the metadata space.

E. Using Cache-Tree to Verify the Correctness of Recovery

STAR restores stale metadata according to the counter-MAC synergization. However, attackers can replace the data, MAC and LSBs with an old tuple to disable the recovery without system detection. For example, a user data with its MAC and the LSBs 0x11 is written into NVM. During recovery, restoring its parent counter block needs the LSB 0x11, but attackers replace the tuple with old data, old MAC and old LSBs 0x10. Due to lazy update in SIT, the root of SIT can't be used to detect the attacks that occur during recovery. Moreover, since the cached tree nodes are lost, they can't be used to detect the attacks on restored metadata. The replay attack also can't be detected via MAC since the old MAC in the user data matches the old data, old LSBs and stale counter in the parent node. Thus the stale parent node is incorrectly restored without detection.

An intuitive approach to detecting the attacks is to construct a merkle tree using the dirty data in the cache [44]. After crashes, if the dirty metadata are incorrectly restored due to attacks, the attacks can be detected by comparing the reconstructed root and the stored one. However, directly constructing the merkle tree faces two challenges: ① the different orders of leaf nodes before and after crashes cause false positives of attacks; and ② inserting/deleting one leaf node may reconstruct the whole tree, thus causing high-overhead recomputation. For the same dataset, the roots of the constructed trees using the data with different leaf-node orders are different. After crashes, the restored metadata are



(a) Set-MAC

(b) Cache-tree

Fig. 9. The MACs in cache lines storing dirty metadata are used to construct a cache-tree.

required to use the same leaf-node order to reconstruct the merkle tree. Otherwise, the attacks would be falsely reported. Thus the order of dirty metadata used to construct the merkle tree needs to be determined. Assuming the dataset is ordered by the ascending addresses of data (descending addresses can also be used). When a cached metadata becomes dirty, the metadata is inserted into the leaves of the merkle tree via its address. The metadata whose addresses are bigger than the inserted metadata need to be moved in the leaf layer as shown in Fig. 8. The ancestor nodes of the moved metadata need to be recomputed with high computation overhead. Deleting the leaf nodes when they become clean also needs to move some leaf nodes and significantly change the merkle tree.

To address the above challenges and verify the correctness of restored metadata, we introduce cache-tree based on the set-way structure of cache to determine the order of dirty metadata and reduce the recomputation overheads of tree nodes. We notice that modern cache is usually a set-way structure. An 8-way cache is divided into multiple sets. Each set has 8 ways, and each way contains one cache line. A specific memory line is cached into a specific set and placed in any way in this set. As shown in Fig. 9, STAR constructs a cache-tree in metadata cache by hashing the MACs of dirty metadata lines to verify the correctness of the recovery. The MACs of the dirty metadata lines in one set are first ordered by the ascending addresses. Then the set-MAC is computed by hashing these ordered MACs of the dirty metadata as shown in Fig. 9(a). Furthermore, a small merkle tree (4 levels in our implementation) is constructed by iteratively hashing these set-MACs as shown in Fig. 9(b), called *cache-tree*. If no dirty metadata lines exist in a set, STAR uses zero-bytes as the set-MAC to construct the cache-tree. During system running, one metadata becoming dirty only affects the set-MAC and changes a branch of the small cache-tree with low computation overhead. After crashes, the restored metadata are ordered in sets via the ascending addresses, which is the same order used before crashes, to construct the cache-tree. STAR logically constructs the cache-tree without physically moving any cache lines. STAR doesn't add an 8-byte space at each set to store set-MAC. Hence the set-MACs and cache-tree nodes exist in metadata cache with SIT nodes and counter blocks. The cache-tree nodes in the cache are not involved in the set-MAC generation. The cache-tree root is always on chip just like a traditional merkle tree root. On recovery, STAR reconstructs the cache-tree. Attacks will cause a wrong MAC and be detected by the cache-tree root.

F. Recovery Process

After system crashes, the stale metadata need to be restored, and the correctness of recovery needs to be verified. To recover the stale security metadata, STAR first reads the multi-layer index from RA to obtain the non-zero L1 bitmap lines. According to the L1 bitmap lines, STAR identifies the stale metadata in metadata space.

STAR recovers the stale SIT nodes and counter blocks in a bottom-up manner. When restoring one stale metadata node, since we don't know which counter in the metadata is dirty, STAR restores all eight counters in the metadata. STAR obtains all LSBs from the eight child nodes' MACs. Coalescing the MSBs in NVM with the obtained LSBs, the eight counters in the stale metadata are restored. The MAC of the stale metadata is re-computed by hashing the eight counters and one corresponding counter in the parent node. To detect the tampering and replay attacks that occur during recovery, STAR caches all MACs of the restored metadata nodes, orders them by the ascending addresses in a set, generates the set-MACs and reconstructs the cache-tree root. The system further verifies the recovery process by matching the recalculated cache-tree root and original on-chip cache-tree root. If the two roots are not matched, attacks occur during recovery and the system recovery fails.

It is worth noting that no matter attacks occur in the recovery-related or recovery-unrelated metadata during recovery, the system has the ability to detect the attacks. Specifically, the recovery-related metadata include the stale security metadata (MSBs in stale nodes), the bitmap lines and the corresponding counters. The tampering and replay attacks in recovery-related metadata result in the wrong set-MACs (i.e., the leaves of cache-tree) in the metadata cache. Thus these attacks are detected by comparing the reconstructed cache-tree root with that stored on chip during recovery. On the other hand, if the recovery-unrelated metadata are attacked, due to not being verified during recovery, the system needs to verify the recovery-unrelated metadata when using these metadata. These attacks will be detected by SIT root or other verified nodes in the cache during running time.

When an attacker replaces the bitmap lines or the security metadata, our STAR can detect that the attack occurs, and the recovery fails. But STAR doesn't know which data block is attacked. It is important to locate the attacked data. However, in existing works, only the strict persistence schemes can locate the attacks with unacceptable write overhead.

IV. Performance Evaluation

A. Evaluation Methodology

To evaluate the performance of STAR, we use Gem5 [8] with NVMain [25] to model the system. NVMain is a cycle-accurate main memory simulator for emerging NVM

TABLE I
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
PCM Main Memory	
Capacity	16GB
PCM latency model	tRCD/tCL/tCWD/tFAW/tWTR/tWR =48/15/13/50/7.5/300 ns
Secure Parameters	
Security metadata cache	512KB, 8-way, 64B Block, in MC
SIT	9 levels, 8-ary, 64B Block
Cache-tree	4 levels, 8-ary, 64B Block
Bitmap lines	1KB, 16 lines, in MC
Multi-layer index	4MB, in NVM

technologies. As illustrated in Table I, we simulate an 8-core X86 processor. The NVM system contains 32KB L1 data and instruction caches, 512KB L2 and 4MB shared L3 caches. Since the structures of SIT node and counter block are identical (i.e., 8 counters and 1 MAC), we use one 512KB metadata cache to store the SIT nodes and counter blocks. The metadata cache is managed by the memory controller [44], [45]. We use 16GB PCM-based main memory, and the PCM latency model is the same as that used in existing works [35], [45]. We implement 5 persistent micro-benchmarks, i.e., array, btree, hash, queue and rbtree, which are widely used in the existing works of persistent memory [7], [11], [17], [18], [23], [27], [45], to evaluate the system. We also leverage macro-benchmarks, i.e., tpcc and ycsb from WHISPER [24] to evaluate the performance of STAR. All these micro- and macro-benchmarks run with 8 threads.

To comprehensively examine the performance of our proposed STAR, we evaluate the following schemes for comparisons.

- A persistent system with a write-back metadata cache (WB). It uses an ideal write-back metadata cache in which only the evicted data from the metadata cache are flushed into NVM. Since not all modified metadata are persistent, the WB scheme doesn't support recovery after system crashes. We use WB as a baseline.
- A strict persistence scheme (Strict Persistence). Strict Persistence scheme persists all changed nodes from the modified counter block up to the root of SIT.
- Anubis for SGX Integrity Tree scheme (Anubis). An ST block, in which Anubis records the address of the dirty metadata, counters and MAC, is written into NVM with each memory write. Anubis recovers the systems according to the ST blocks.
- Our proposed STAR. STAR leverages counter-MAC synergization to persist the modifications of nodes, while providing atomicity guarantee. The bitmap lines and multi-layer index ensure the fast recovery of security metadata after crashes.

It is worth noting that Osiris and Triad-NVM can't be used to recover the counter blocks and integrity tree nodes in SIT-based persistent memory, and we don't compare our STAR with them.

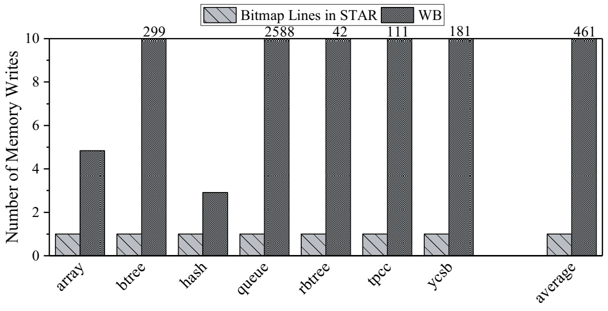


Fig. 10. The write number of Bitmap Lines in STAR compared with that of WB (normalized to Bitmap Lines in STAR).

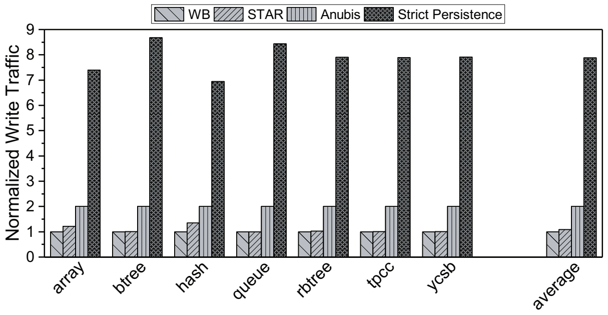


Fig. 11. The write traffic of different schemes (normalized to WB).

B. Write traffic of different schemes

The memory writes of STAR consist of bitmap line writes and normal memory line writes. We show the write overhead of the bitmap lines of STAR, and compare all write traffic of STAR with other schemes.

Fig. 10 shows the number of bitmap lines writes of STAR compared with the number of WB writes. The number of bitmap lines in ADR is 16, consisting of 2 L2 bitmap lines and 14 L1 bitmap lines. STAR flushes the bitmap lines to NVM since there is a bitmap line miss. The bitmap lines selected by LRU need to be flushed into the recovery area (RA) in NVM, and one bitmap line needs to be read from RA to record the locations of dirty metadata. Fig. 10 shows that the number of the bitmap lines writes/reads in STAR is negligible compared with WB. On average, the number of WB writes is 461x more than that of bitmap lines writes. Because one bitmap line covers 8-page metadata space (512×64B=32KB). When the applications have high spatial localities, STAR rarely evicts bitmap lines. Most bitmap lines writes are caused by evicting security metadata from metadata cache since the security metadata have lower spatial locality than user data. Fig. 10 shows that in different workloads, the numbers of bitmap lines writes become different, which depend on the localities of workloads.

Fig. 11 shows the write traffic of different schemes. In addition to persisting common memory writes (i.e., the writes in WB), Anubis also persists ST blocks, and the strict persistence persists all nodes in a branch of SIT when a user data is written. Since the height of SIT in 16GB persistent memory is 8 (excepting the root), the write traffic (including persisting the user data) of a strict persistence scheme is 9

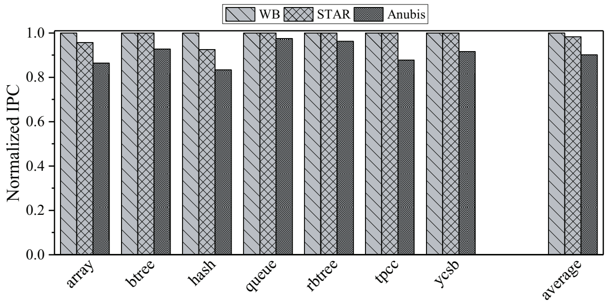


Fig. 12. The IPCs of different schemes on different workloads (normalized to WB).

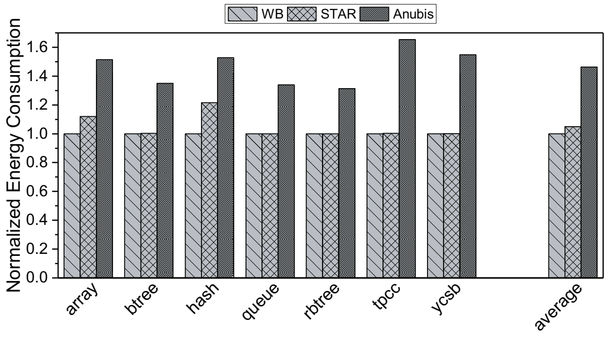


Fig. 13. The energy consumption on different workloads (normalized to WB).

times that of WB in theory. However, in Fig. 11, the write traffic of the strict persistence scheme is less than 9 times that of WB, since the tree nodes also need to be evicted in the WB scheme according to cache replacement policy, i.e., WB actually has more writes. Compared with the baseline WB scheme, the write traffic of STAR is 1.08x, while Anubis has 2x write traffic than WB. STAR significantly reduces 92% extra memory traffic compared with Anubis. Extra write traffic in STAR is caused by writing bitmap lines.

C. The IPCs

Due to low memory writes, the performance overhead of STAR is low. Fig. 12 shows the IPCs (instructions per cycle) of different schemes. Since the array and hash have more memory writes in STAR (about 1.21x and 1.34x memory writes than that of WB), they have more IPC degradation. However, the highest IPC overhead in the hash workload is 8%, which is light and acceptable. We also consider the performance impact on cache-tree. The performance overhead incurred by cache-tree is negligible. Since the metadata cache is small (512KB), the height of the cache-tree is low (4 levels). Moreover, we don't force to persist cache-tree node. After system crashes, we reconstruct the small cache-tree from leaf nodes. The leaf nodes, i.e., the dirty cached metadata, can be restored via the counter-MAC synergization after crashes. From Fig. 12, STAR has a better IPC than Anubis in all workloads. On average, STAR achieves about 98% IPC compared with WB, and Anubis achieves about 90% IPC.

D. The Energy Consumption

NVMs have asymmetric read and write energy consumption, and the write energy consumption is high. By reducing the memory writes to NVM, STAR significantly reduces the energy consumption of the NVM systems. Fig. 13 shows the energy consumption on WB, STAR and Anubis. Like the observation in IPC performance, since the array and hash incur more writes, they consume more energy compared with other workloads. On average, STAR significantly reduces the energy overhead from 46% in Anubis to 4%.

TABLE II
THE HIT RATIOS OF DIFFERENT NUMBERS OF BITMAP LINES PLACED IN ADR (2, 4, 8, 16 AND 32).

Bitmap Lines	2	4	8	16	32
Hit Ratio	32.85%	47.44%	64.37%	74.75%	82.19%

E. The Sensitivity to the Number of Bitmap Lines in ADR

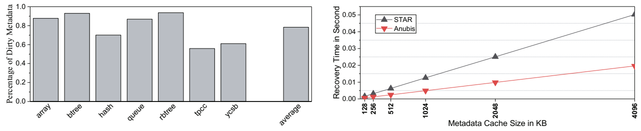
A bitmap line in ADR covers 8 memory pages since one bitmap line contains 512 bits and one bit represents one memory line. More bitmap lines in ADR cover more metadata space, thus increasing the hit ratio of the bitmap lines. Table II shows the average hit ratio among 2, 4, 8, 16 and 32 bitmap lines in ADR. The hit ratio is not too high because the system tries to access the bitmap lines only when the dirty state of one metadata line changes. For example, a user data is written into NVM, which results in its counter block to become dirty. The location of this counter block needs to be recorded in the bitmap lines. Furthermore, the neighbor user data with the same counter block is written. Since the counter block has become dirty and the state of the counter block is not changed, the system doesn't need to access the bitmap lines and record this location again. When the dirty metadata is evicted from cache, the system records its location in a bitmap line due to its state changing from dirty to clean. But when a clean metadata is evicted, the system will not access the bitmap lines. Table II shows that more bitmap lines in ADR provide a higher hit ratio, which causes less number of bitmap lines to be written. Considering that the on-chip ADR region is expensive, and the improvement upon hit ratio decreases with more bitmap lines, we choose to place 16 bitmap lines in ADR.

F. Recovery Time

To recover the stale nodes, STAR first reads the non-zero L1 bitmap lines according to the multi-layer index to obtain the locations of the stale nodes. STAR further recovers a stale node by reading its parent node and eight child nodes to restore the counters and MAC of the stale node. The cache-tree is further reconstructed to verify the correctness of the recovery process. Like Anubis and Osiris [40], [44], we assume that fetching and updating one metadata (64 bytes) from NVM consume 100ns. The recovery time consists of fetching metadata from memory and reconstructing the cache-tree. Since the cache-tree is small, the latencies of reading metadata from NVM dominate the recovery time.

Fig. 14(a) shows the percentage of dirty metadata in the metadata cache. When system crashes occur, these dirty metadata are stale in NVM and need to be restored. STAR hence needs to restore 78% metadata of cache, much smaller than 100% in Anubis.

During the recovery process, STAR reads the negligible number of bitmap lines and restored metadata to verify the



(a) Dirty metadata ratio

(b) Recovery time

Fig. 14. The recovery time of STAR and Anubis for recovering dirty metadata in different sizes of metadata cache.

recovery process. Restoring each stale metadata needs to read 10 related nodes, including 1 stale node to be restored, 1 parent node and 8 child nodes.

Fig. 14(b) shows the recovery time after system crashes of different schemes. For a 4MB metadata cache, STAR needs 0.05s to recover the stale security metadata, while Anubis needs 0.02s. Although STAR needs about 2.5x recovery time than Anubis, STAR also requires less than 0.1s to recover a sufficiently big (4MB) cache for the security metadata. Moreover, since the systems need 10–100s to execute self-test after system crashes [4], our time for recovering stale security metadata is negligible in the real-world systems.

G. The Benefits of Different Optimizations

Counter-MAC synergization and bitmap lines disaggregate contents and addresses of the modified metadata, allowing each to be independently persisted and thus eliminating unnecessary memory writes. They reduce 92% extra write traffic. Due to the recorded addresses, STAR only recovers the stale metadata, and proposes the multi-layer index to limit the metadata recovery time to less than 0.1s. Finally, the cache-tree is used to detect the attacks that occur during recovery.

V. Related Work

Recovery in NVM. Fast and cost-efficient recovery in NVM is important to secure persistent memory systems. Osiris [40] recovers the counter blocks by retrieving counter from the *stale counter* to *stale counter+N* and leverages error-correction codes to verify the correctness of the retrieved counter blocks. Since Osiris writes a counter block into NVM when one counter in this block has been increased by N times, Osiris has fewer writes than write-back schemes. cc-NVM [38] caches the flushed counter blocks in write pending queue with a battery in an epoch and only flushes these blocks at the end of one epoch. cc-NVM also retrieves the counter to obtain the correct one. Unlike Osiris, cc-NVM uses MAC to verify the correctness of counters. To recover a merkle tree, Triad-NVM [7] flushes the N lowest levels nodes and counter blocks with the user data writes. On recovery, Triad-NVM reconstructs the whole tree from the flushed tree nodes instead of user data. Anubis [44] provides a fast recovery scheme for merkle tree and SGX integrity tree by recording the addresses of the modified metadata in a shadow table block that is flushed with user data. Anubis only recovers the cached nodes according to shadow table blocks and reduces the recovery time. Phoenix [5] combines the Osiris and Anubis to reduce the write overhead of metadata. Phoenix relaxes the persistences of counter blocks using Osiris, and decreases the extra writes of shadow table blocks for counter blocks.

Secure NVM. NVM suffers from the data remanence vulnerability and limited lifetime. Ensuring data confidentiality with low write overhead in NVM is necessary and important. DEUCE [41] proposes a dual-counter scheme. In one epoch, DEUCE uses the old counter to encrypt the untouched words and new counter to encrypt the changed words in a cache line, which reduces the write traffic since the untouched words needn't to be written by executing DCW [37] and FNW [10]. Based on DEUCE, SECRET [33] further reduces the zero-content words writes in NVM by flushing a zero-flag. Silent Shredder [6] observes that zeroing out physical pages before mapping to processor consumes a large percentage of memory writes. They initialize the counters of the pages with low overhead. The initialized pages can be assigned without zeroing out. SuperMem [45] uses a write-through counter cache to ensure the counter crash consistency and proposes a counter write coalescing (CWC) scheme to reduce the number of counter writes. Freij et al. [13] point out that prior research under-estimated the cost of updating BMT, and propose the pipelining BMT updates scheme to reduce the update latency of BMT.

Unlike existing schemes, STAR focuses on SIT lazy scheme and instantly persists the modifications in the cache to achieve low recovery and write overheads.

VI. Conclusion

This paper proposes STAR to reduce the recovery time and write overhead of recovering the SGX integrity tree nodes and counter blocks in the secure non-volatile memories. To efficiently restore the dirty metadata and verify the recovery process, STAR judiciously exploits the unused space in MAC in one node to store the LSBs of the corresponding counter in the parent node, and leverages bitmap lines in ADR to maintain the locations of stale metadata. Moreover, a multi-layer index is used to speed up recovering stale nodes, and a cache-tree is constructed to ensure the correctness of the recovery process. Experimental results show that compared with state-of-the-art work, STAR significantly reduces the write overhead and improves system performance while fast recovering the security metadata after crashes.

Acknowledgements

This work was supported in part by National Key Research and Development Program of China under Grant 2016YFB1000202, National Natural Science Foundation of China (NSFC) under Grant No. 61772212 and Key Laboratory of Information Storage System, Ministry of Education of China.

References

[1] "Amazon.com goes down, loses $66,240 per minute," https://www.forbes.com/sites/kellyclay/2013/08/19/amazon-com-goes-down-loses-66240-per-minute/#7e75d897495c, accessed July 20, 2020.

[2] "The cost of it downtime," https://www.the20.com/blog/the-cost-of-it-downtime/, accessed July 20, 2019.

[3] "Intel corporation. persistent memory programming." https://pmem.io/.

[4] "Power-on self-test," https://en.wikipedia.org/wiki/Power-on_self-test, accessed July 20, 2019.

[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*, 2016.

[7] 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.

[8] 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, 2011.

[9] 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.

[10] S. Cho and H. Lee, "Flip-n-write: a simple deterministic technique to improve pram write performance, energy and endurance," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2009, pp. 347–357.

[11] 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*, 2012.

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

[13] 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.

[14] A. Gamatié, A. Nocua, J. Weloli, G. Sassatelli, L. Torres, D. Novo, and M. Robert, "Emerging nvm technologies in main memory for energy-efficient hpc: an empirical study," 2019.

[15] 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.

[16] A. Hassan, H. Vandierendonck, and D. S. Nikolopoulos, "Software-managed energy-efficient hybrid dram/nvm main memory," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, 2015.

[17] 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.

[18] 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*, 2016.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with WHISPER," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] S. Schechter, G. H. Loh, K. Strauss, and D. Burger, "Use ecp, not ecc, for hard failures in resistive memories," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 141–152.

[32] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for nvm," in *50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017, pp. 178–190.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] B.-D. Yang, J.-E. Lee, J.-S. Kim, J. Cho, S.-Y. Lee, and B.-G. Yu, "A low power phase-change random access memory using a data-comparison write scheme," in *IEEE International Symposium on Circuits and Systems*. IEEE, 2007, pp. 3014–3017.

[38] F. Yang, Y. Lu, Y. Chen, H. Mao, and J. Shu, "No compromises: Secure nvm with crash consistency, write-efficiency and high-performance," in *Proceedings of the 56th Annual Design Automation Conference (DAC)*.

[39] J. J. Yang, D. B. Strukov, and D. R. Stewart, "Memristive devices for computing," *Nature nanotechnology*, vol. 8, no. 1, p. 13, 2013.

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

[41] 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.

[42] J. Yue and Y. Zhu, "Accelerating write by exploiting pcm asymmetries," in *IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2013, pp. 282–293.

[43] 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. ACM, 2009.

[44] 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.

[45] 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.

[46] 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.