CacheTree: Reducing Integrity Verification Overhead of Secure Non-Volatile Memories

Zhengguo Chen*, Youtao Zhang† and Nong Xiao‡
*State Key Laboratory of High Performance Computing, School of Computer, National University of Defense Technology
†Department of Computer Science, University of Pittsburgh, Pittsburgh, USA
‡School of Data and Computer Science, Sun Yat-sen University, Guangzhou, China
Email: zgchen.nudt@foxmail.com, zhangyt@cs.pitt.edu, xiaon6@mail.sysu.edu.cn

Abstract—Emerging non-volatile memories (NVMs), while exhibiting great potential to be DRAM alternatives, are vulnerable to security attacks. Secure NVM designs demand data persistence on top of traditional confidentiality and integrity protection. A simple adoption of existing secure memory designs would incur non-negligible overheads, including performance degradation, NVM lifetime reduction and energy consumption increase.

In this paper, we propose CacheTree to address the integrity verification overhead for secure NVMs. By constructing extra Merkle trees on top of metadata cache, CacheTree helps to authenticate the volatile cache contents, which enables the adoption of write-back policy and prevents frequent NVM writes in persisting metadata. We then adopt CacheTree to address the integrity verification in secure NVM, in particular, the overheads in persisting MACs (for protecting the integrity of user data at memory line level) and persisting the main Merkle tree (for protecting the integrity of the whole memory space). Our experimental results show that CacheTree, with less than 0.5% storage overhead, achieves up to 20.1% performance improvement, 44.3% lifetime increase and 43.7% energy consumption reduction over the state-of-the-art solutions.

Index Terms—CacheTree, security, non-volatile memories, integrity.

I. INTRODUCTION

Emerging non-volatile memories (NVMs), e.g., PCM and ReRAM [15], [30], [36], [42], exhibit great potential to be DRAM alternatives in future computers. NVM based memory subsystems, in addition to having density and scalability advantages, can effectively support data persistence, allowing fast recovery from system crashes or power failures. However, such memory systems are vulnerable to security attacks [33], including confidentiality attacks such as bus snooping and memory scanning attacks [5], [8], [40] and integrity attacks such as splicing, spoofing and replay attacks [3], [23].

Traditional DRAM-based secure memory adopts counter mode AES algorithms, e.g., Galois Counter Mode (GCM) [19], [37], to achieve high performance data encryption and authentication. The memory is partitioned to four regions that hold ciphertext, counters, MACs, and Merkle tree nodes, respectively. While the ciphertext region saves the encrypted data, the counter region saves the cleartext counter values to support GCM encryption. A cryptographic signature, referred to as message authentication code (MAC), is generated for each cacheline to ensure the data integrity at the block level, while a Merkle Tree (MT) is then built for the whole memory space to ensure the overall data integrity [12]. The data other than ciphertext are referred to as security metadata.

Many optimization schemes have been proposed to reduce integrity verification overhead. Rogers et al. proposed to adopt Bonsai Merkle Tree (BMT) [25] that builds the merkle tree on counters instead of MACs, leading to significant tree size and verification overhead reduction. Rakshit et al. [23] proposed to create an extra small MT for the frequently accessed memory data and keeps its root in on-chip secure register to reduce authentication overhead.

Constructing secure NVM needs to enforce data persistence over security protection. All the updates to security metadata need to be persisted along with the updates to the ciphertext [17], [24], [39], [45]. Given the non-trivial metadata cache, it is impractical to install battery or super capacitors to flush all buffered data at system crash time [4], [44]. A simple solution to implement secure NVM is to persist security metadata to NVM at each write, which introduces tens of times NVM writes and thus degrades both the system performance and the NVM chip lifetime. Osiris [39] successfully integrates the write back policy in persisting counters. Anubis [44] mitigates the recovery overhead but increases the number of NVM writes as it records metacache update in NVM. In summary, it remains challenging to achieve high performance security protection with data persistence.

In this paper, we propose CacheTree for secure NVMs. By constructing extra Merkle trees for dirty entries in metadata caches, CacheTree enables the adoption of the write-back policy rather than the write-through policy for metadata caches, we effectively address integrity verification overhead for secure NVMs.

Our contributions are as follows:
- By constructing extra Merkle trees on top of metadata cache, CacheTree helps to authenticate the volatile cache contents, which enables the adoption of write-back policy and prevents frequent NVM writes in persisting metadata.
- We present two use case studies — MACTree and HNodeTree. MacTree creates an extra Merkle tree on MAC cache to effectively reduce the number of NVM writes in persisting MACs; HNodeTree creates an extra Merkle tree on dynamically chosen nodes of the main Merkle tree, which
reduces the number of Merkle tree nodes to be checked and updated when authenticating memory accesses.

- We evaluate the design and compare it with the state-of-the-art solutions. Our experimental results show that CacheTree, with less than 0.5% storage overhead, achieves up to 20.1% performance improvement, 44.3% lifetime increase and 43.7% energy consumption reduction over the state-of-the-art solutions.

II. BACKGROUND

A. Attack Model

Before designing the secure NVM system, we first define the trusted computing base (TCB) and the threat model. Similar to that in traditional trusted computing, our TCB includes the processor only, i.e., on-chip components are trustworthy while off-chip components, such as memory buses and memory modules, are vulnerable to security attacks. We assume the OS is not trustworthy.

In this paper, we are to defend against three types of security attacks, similar to those defended in DRAM-based secure memory schemes [12], [25], [28], [37]: (1) we are to encrypt the user data to protect data confidentiality, i.e., to prevent attackers from revealing the cleartext; (2) we are to authenticate the user data to prevent attackers from compromising the data; (3) we are to check the overall data integrity to prevent data splicing, spoofing, and replay attacks.

Fig. 1: The counter mode encryption and authentication.

B. Counter-mode Encryption

Existing secure memory designs widely adopt counter mode AES encryption and authentication, in particular, Galois Counter Mode (GCM) algorithms [19]. Figure 1(a) shows the procedure of counter mode encryption. We assume the encryption is at the 64B memory line granularity. To encrypt, the system generates a one-time pad (OTP) by applying AES encryption on a seed that includes the memory address, a counter value, and an encryption initialization vector (EIV). The system then generates the ciphertext (plaintext) by XOR-ing the OTP with the plaintext (ciphertext) for encryption (decryption). A 64-bit counter is assigned to each memory line so that every update increments the counter, which avoids using the same OTP before and after each write. A counter overflow incurs large overhead as we need to change the EIV and re-encrypt all memory lines. To mitigate counter space and overflow overheads, Yan et al. proposed the split-counter design [37] to group 64 counters in one memory line, which includes a 64-bit shared major counter and 64 7-bit minor counters, shown in Figure 1(a). If a minor counter overflows, the system increments the major counter, resets all the minor counters, and re-encrypts all 64 memory lines in the group.

C. Integrity Verification

To prevent malicious modification of memory content, the system creates a message authentication code (MAC) for each cacheline. As shown in Figure 1(b), the MAC is generated from the ciphertext together with the data address, the counter and an initialized vector (AIV), using a hash function, e.g., AES-based GHASH function [19], [37]. A MAC is usually 128 bit long [37]. For each memory access, the system needs to recompute the MAC and verify it with the stored MAC. Any malicious modification to data, counter or MAC shall lead to a MAC mismatch, hence the detection.

To defend data replay attacks, the system builds a Merkle tree (MT) on all MACs with cryptographic hash, and saves the root hash in an on-chip secure register. A memory read access needs to recompute the root hash and compare against the saved one to verify integrity [3], [23]. Given that the verification overhead is linear to the height of the tree, Rogers et al. proposed the BMT (Bonsai Merkle Tree) design [25] that builds the tree on counters rather than MACs.

Fig. 2: An overview of the BMT.

Figure 2 illustrates how a BMT works with the split-counter scheme [6], [39], [44]. At the leaf node level, each memory line contains 64 counters. With each memory line producing an 8B HMAC (hash MAC), the 8-ary BMT saves eight HMACs in each internal MT node. The root hash is kept in a secure on-chip register. A BMT verification (or update) involves checking (or updating) all nodes along a path. While the verification can be done in parallel, the update needs to be in serial from bottom to top, as the upper-level HMACs depend on contents of the lower-level nodes.

Fig. 3: An overview of the SGX-style tree.

Another popular Merkle tree design, referred to as SGX-style tree in this paper, was designed to process the verification (and the update) in parallel [9], [13]. As shown in Figure 3, an SGX-style tree keeps counters in both the leaf nodes and the internal nodes. When incrementing a counter in the child node, we increment its parent node accordingly and repeat the process until the root. Each node also keeps an HMAC that is generated from the parent’s as well as the children’s counters.
While a memory write needs to increment all nodes along its authentication path, these updates can be done in parallel. The SGX-style MT was widely adopted in recent studies such as Synergy [28], VAULT [34] and Morphable Counters [27].

D. Secure NVM Access

Constructing secure NVM demands data persistence on top of security protection, which significantly complicates the design. As memory writes need to update both user data and security metadata, a simple approach is to adopt write-through policy for metadata caches, and persist everything to NVM, including ciphertext, counter, MAC and MT nodes, as shown in Figure 4. That is, a write operation may commit only after updating ciphertext and metadata in both cache and NVM. Clearly, such an approach introduces large performance overhead and degrades NVM chip lifetime dramatically.

In Section II-C, we show that there are two types of Merkle trees, i.e., BMT and SGX-style trees. We next elaborate that the persistence requirements are different for the two types.

Since a BMT tree is built on top of counters, its internal nodes are generated hashes, and the root hash is kept in an on-chip persistent register, it is unnecessary to persist its internal nodes to NVM [39], [44]. That is, keeping some internal nodes in MT node cache is just to speed up the verification process for reads. As long as the counters are persisted in NVM, all the internal nodes can be reconstructed after a system crash, with the integrity being checked against the root hash [33]. However, an SGX-style tree needs to persist internal MT nodes to support data persistence. This is because its internal nodes keep derivative counters, which cannot be reconstructed if lost during a system crash. As a result, we need to persist all the nodes along the path, as shown in Figure 4.

Servicing memory reads in secure NVM is similar to that in secure DRAM-based memory, which includes fetching and checking all nodes on the path from the leaf to the root. To mitigate the overhead, the memory fetches happen in parallel, while secure hashes, e.g., MAC and HMAC, are processed using a pipeline AES implementation [5], [39].

Recent studies mitigate the update overhead in secure NVM [17], [38], [39], [45]. Given the counter mode encryption and decryption achieves high performance, the studies focus mainly on mitigating the update of security metadata. Osiris [39] successfully integrates the write back policy in persisting counters. Anubis [44] mitigates the recovery overhead by recording the updates to metadata cache in NVM. Unfortunately, it remains challenging to achieve high performance security protection with data persistence.

III. CacheTree Design Principles

In this section, we present CacheTree, a mechanism that builds an extra Merkle tree on security metadata cache. The design goal is to enable the adoption of write-back cache replacement policy for the metadata caches. We elaborate its design principles in this section and employ it to address the design challenges in secure NVM in the following section.

Figure 5 shows how CacheTree works. Given a four-way set-associative security metadata cache $C$, we assume one cache set contains three cache blocks $B_0$, $B_1$ and $B_2$ and one empty cache entry. As we discussed, a simple approach to achieving persistence in secure NVM is to adopt write-through cache replacement policy — an update not only updates the copy in the cache but also the copy in the NVM, as shown in Figure 5(a). When the system crashes, we do not need to recover the security metacache as the NVM keeps the most up-to-date copy. The limitation of such an approach is that it increases the number of NVM writes.

As a comparison, adopting write-back policy cannot ensure persistence. For example, assume at an update time, we only update the copy in the metacache, e.g., blocks $B_0$ and $B_1$. When the system crashes, since the cache is a security metacache, we need to recover these blocks to authenticate the ciphertext. We may find a mismatch of the stale copy of $B_0$ and its associated ciphertext, however, we cannot distinguish it from the mismatch that an attacker maliciously modifies $B_2$.

In this paper, we propose CacheTree to create extra Merkle trees (MTs) on metacaches such that we can support persistence with write-back policy. The extra Merkle tree is not a SGX-style MT. However, it is similar to BMT, built by iteratively hashing from the leaf nodes to Root. Similar as that in previous studies [39], [44], their root hashes are kept in secure on-chip registers. However, the most notable difference between CacheTree and the MTs in existing schemes is that CacheTree is built on cache contents while previous MTs are built on memory lines.

It is challenging to build an MT on cache contents as the contents are lost during a system crash. For a set-associative cache, a large number of memory lines may map to any physical location in a cache set. To ensure the deterministic re-generation of the MT root for authentication, we need to determine not only the contents (i.e., the memory lines) involved in generating the old root, but also their orders. We next discuss the three critical design issues in CacheTree.

- **Determine what memory lines/cache blocks are involved.** The first design issue is to determine what memory lines are involved in generating the root hash. Simple approaches includes tracking all addresses in NVM and/or mapping the cache contents to NVM (as that in Anubis [44]). However, these approaches would introduce extra NVM writes that we are to avoid. In this paper, we devise to use the dirty cache entries only, which enables the identification of involved memory lines without tracking overhead. That is, since the security metacache is to authenticate the ciphertext, a dirty entry in the metacache indicates a mismatch of its ciphertext and the stale copy of security metadata in NVM.
• **Determine the contents of the involved cache blocks.** The next design issue is to track the cache contents used in generating the root hash. Given persisting dirty entries in NVM increases NVM writes, in this paper, we exploit the characteristic of security metacache, i.e., it is to authenticate the ciphertext while the ciphertext in NVM is always up-to-date. We therefore regenerate the metacache contents based on the ciphertext of the identified memory lines.

• **Determine the order of involved cache blocks.** Given a fixed set of cache blocks, we would generate different Merkle tree roots if computing with different orders. Since a memory line may stay in any entry in a cache set, and its exact location information will be lost in a system crash, we decide to choose a fixed order in computing the root hash. This paper uses the following rules to order: (i) dirty entries are ordered using their tag values; (ii) clean and empty cache entries are placed as dummy block (all 0s) at the end of the set. In Figure 5(c), assume $B_0$ and $B_1$ are dirty entries, $B_2$ is clean, and Tag($B_2$) = 0x100, Tag($B_1$) = 0x011, Tag($B_0$) = 0x100, then we have “$B_1$,$B_0$,0”. Note, $B_2$ is converted to all 0s as it is a clean entry.

Next, we discuss how to exploit CacheTree to recover the cached contents of metadata and detect attacks if there is any. We focus on metadata as the ciphertext has been persisted. When we adopt the write back cache replacement policy, there exists five types of metadata: (D1) the clean metadata; (D2) the memory locations of the dirty metadata in meta cache; (D3) the contents of the dirty metadata in meta cache; (D4) the HMACs of our CacheTree for (D3); and (D5) the root hash. From above discussion, before committing a write operation, CacheTree persists (D5) in secure register similar as that in existing secure memory designs. (D2) is identified through mismatched updated ciphertext and stale metadata. In the case of a system crash, we may lose (D3) and/or (D4). We first generate (D3) based on (D2), i.e., $B_0$ and $B_1$ would be identified and re-generated in the example, and then generate (D4) according to our ordering rules. This produces a new root hash. A mismatch of the new root hash and the saved (D5) would alarm a security attack. Otherwise, the metadata is successfully recovered so that the system can resume. A system crash can’t mess up (D2) as it is persisted before committing any write. For a security attack that alters (D2), (D3), or (D4) would lead to a mismatch of (D5).

In summary, CacheTree enables the adoption of write-back policy in persisting security metadata, which leads to improvements on both performance and chip lifetime.

IV. CASE STUDY 1: THE MACTree DESIGN

CacheTree is a powerful authentication mechanism that has many applications. In this paper, we employ it to address the authentication overhead in secure NVM.

In our baseline secure NVM design, we integrate BMT with Osiris. Choosing Osiris enables the write-back policy for the counter cache. Choosing BMT helps to avoid persisting the internal nodes of the Merkle tree. We will elaborate this choice in the next section. While this baseline design significantly optimizes the naive approach that persists everything, it still faces large performance degradation over the secure memory design, i.e., the one with no data persistence demand.

A. Persisting MAC is Expensive

Figure 6 compares the numbers of NVM writes and the performance, respectively, when adopting cache-back or write-through replacement policies. From the figure, adopting write-through policy introduces on average 80% more NVM writes and 11.4% performance degradation, indicating that it is important to mitigate the overhead in persisting MACs.

While it is possible to persist critical data (e.g., ciphertext and MACs) in NV cache [43], an NV-cache based design tends to incur larger overheads, making it less preferable for...
immediate integration in modern computer systems. Compared to a traditional SRAM cache, an NV cache has longer write latency and larger energy consumption [41]. Given a user data update for secure NVM results in multiple writes to update user data and security metadata, respectively, an NV-cache based design shall incur large performance, lifetime and energy consumption overheads, as shown in the experiments section. Another solution is to persist MACs in parallel with ciphertext using extra memory channels [28] and/or ECC chip, e.g., Intel Optane memory module consists of 11 chips [1]. However, NVM tends to suffer from high soft- and hard-errors, making it critical to adopt strong ECC that uses more ECC chips for extended NVM chip lifetime. Exploiting extra memory channels tends to degrade system throughput. To reduce MAC overhead, Synergy [28] stores MACs in the ECC area, which can be written together with ciphertext. While it helps to improve read performance by fetching ciphertext and MAC in one read, Synergy creates an extra parity to ensure error correction capability. Thus, it still need two NVM writes to persist ciphertext, ECC, and MAC.

writes, we adopt Asynchronous DRAM Self-Refresh (ADR) and Write-Pending-Queue (WPQ) techniques such that the memory controller has enough power to flush the contents of WPQ to NVM and guarantee the atomic update [11], [26], [29], [44]. We create two extra CacheTrees — MACTree and HNodeTree, as shown in the figure. Given the extra Merkle trees are small, we keep all their nodes on-chip.

1: updating the counter cache using Osiris [39].
2: updating the MAC cache using MACTree.
3: persisting the ciphertext to the NVM.
4: updating the main Merkle tree using HNodeTree.

C. The Design of MACTree

The MACTree scheme is a direct application the CacheTree design. We create an extra MT on dirty MAC cache entries and save its root hash $R_1$ in a secure on-chip register.

Regarding the three design issues of CacheTree, we choose to (1) only use dirty MAC cache entries; (2) sort the cache entries before hashing; (3) regenerating the MACs from ciphertext in case of system crashes.

When inserting a new cacheline in the cache or updating a clean entry, we sort all entries in the corresponding cache set. Given this order is not changed when we update entry contents only or read the contents for authentication, we keep an $m$-bit flag for each cacheline (of $2^m$-way set associative cache) to record its sorted location. When updating the MAC cache, we may need to evict a cacheline to place the new line. We need to persist the evicted dirty line before committing the write operation.

While MACTree maintains an extra Merkle tree, the overhead is usually small. For example, for a 32KB 4-way set associative MAC cache, we only need to build a 4-level 8-ary Merkle tree. We adopt a pipeline implementation of AES engine such that we generate one HMAC per cycle, and overlap the computation of multiple HMAC hashes with existing HMAC computations of the major BMT.

D. MAC Recovery

We next briefly discuss the recovery using MACTree. Since we adopt write-back policy for the MAC cache, its contents will be lost if the system crashes. To recover its contents,

Step 1: CacheTree first utilizes Osiris to recover the counters in NVM if there any mismatches [39].

Step 2: It computes all MACs (MAC$_{comp}$) based on the ciphertext and the counters, and matches the computed ones with the saved MAC copies (MAC$_{old}$) in NVM.

Step 3: Assuming there is no malicious attack, a MAC mismatch indicates that the MAC$_{comp}$ is the non-persisted dirty MAC in the MAC cache before crash. Therefore, at this step, we collect all mismatched MAC lines (64B) and placed them to the MAC cache according to their memory addresses. We place the computed MAC MAC$_{comp}$ in the MAC cache and mark them as dirty.

If multiple lines are brought to one cache set, we place them in sorted order (based on their memory addresses). The number of lines mapped to one cache set must not be

Fig. 7: An overview of CacheTree.

B. A High Level Overview of Our Designs

Figure 7 presents an overview of our designs in this paper. The blue blocks are the traditional security enhancement components, including AES pipeline engine, MAC cache, counter cache, Merkle tree cache and a secure non-volatile register to save the root hash $R_0$ of the Merkle tree. The orange blocks are those we added to enable CacheTree. All on-chip components are trustworthy.

For a memory read that misses in the last level cache, the memory controller accesses its ciphertext from the NVM and accesses its corresponding metadata (counter, MAC, and MT nodes) from the three metadata caches, respectively. Any miss results in additional NVM read to fetch the needed data. The memory controller then verifies the integrity by computing and matching MAC and HMAC, decrypts the ciphertext, and returns the requested plaintext.

For a memory write, the memory controller first checks if the data is in the LLC and, if not, completes the integrity verification and decryption in the same way as that in servicing a read. It then updates the security metadata, and re-encrypts and updates the requested data. As shown in Figure 7, it consists of four updates as follows. For its associated NVM
bigger than the number of cache ways. Otherwise, we stop the recovery and alarm the attack.

Step ②: We rebuild the extra Merkle tree using the contents in the MAC cache. Empty cachelines are treated as dummy ones (all 0s).

Step ⑤: We continuously compute the HMACs on MT internal nodes till we compute the MACTree root hash \( R_{\text{comp}} \). If \( R_{\text{comp}} \) matches the MACTree root \( R_1 \) stored in the secure on-chip register, we successfully reconstruct the MAC cache and the extra Merkle tree.

All non-empty cachelines in the MAC cache are marked as dirty. They shall be persisted in NVM at a later eviction time.

Step ⑥: We then re-construct and authenticate the main BMT Merkle tree. We elaborate the details in the next section.

E. Security Analysis

Given MACTree is built on top of BMT and Osiris, their security protection mechanisms help to detect malicious changes made to the counters and/or the main Merkle tree.

If there are malicious changes made to the ciphertext or the MAC, we would detect MAC mismatches, which are in addition to the mismatches on non-persisted dirty MACs. Since the latter are the only ones used to produce the saved root hash \( R_1 \), having the former mismatches involved in constructing the extra Merkle tree would lead to mismatch of the root hash, i.e., the detection of the attack.

If an attacker can make malicious changes without causing MAC mismatch, the root hash \( R_1 \) would be a match, i.e., there is no detection. However, such an attack would also succeed in the baseline secure memory design, i.e., the attacker can generate two memory blocks that generate the same MAC. The possibility is extremely low, as shown in [16], [25].

V. CASE STUDY 2: THE HNODETREE DESIGN

In this section, we employ the CacheTree design to mitigate the overhead in authenticating the main Merkle tree.

A. The Selection of Merkle Tree for Secure NVM

Given there are two types of Merkle trees, we first determine the appropriate one for secure NVMs. Anubis [44] considers the persistence and security of NVM systems and proposes AGIT and ASIT schemes for BMT and SGX-style MTs, respectively. During memory write, it shadows metadata cache information in a shadow table (ST) residing in NVM, which helps to recover system rapidly. For counter and MT nodes, AGIT incurs small NVM write overhead since it only shadows the addresses of dirty metadata cachelines. However, ASIT incurs one extra NVM write per memory write because it needs to shadow both addresses and the dirty content [44].

Figure 8 compares the number of NVM writes when using a BMT tree (BMT), a BMT enhanced by AGIT (AGIT), a SGX tree (SGX), and a SGX tree enhanced by ASIT [44] (ASIT). From the figure, we observe that ASIT increases about 8% more NVM writes than BMT. While SGX has on average 3.2× more NVM writes than BMT, ASIT greatly reduces the extra NVM writes to 15.9% on average over BMT.

B. HMAC Computation and Update in BMT is Expensive

Given most NVMs have limited lifetime, it is beneficial to adopt a Merkle tree that leads to fewer NVM writes. Adopting Anubis helps to improve recovery time but suffers from more NVM writes than the baseline. This is because Anubis needs to record the extra metadata in NVM, which introduces more NVM write. Therefore, we choose BMT in our baseline.

C. The Design of HNODETREE

Figure 10(a) illustrates how HNODETREE works. The right tree represents the main Merkle tree. At runtime, we dynamically choose a subset of frequently accessed tree nodes, e.g., \( H_0 \) and \( H_7 \), and place them in a newly added HNODE cache. We then construct an extra Merkle tree on the HNODE cache and compute its root hash \( R_2 \). While \( R_2 \) is saved in a secure on-chip register, other nodes of the extra Merkle tree are saved in volatile SRAM buffer.

To service a write, assume we have persisted the ciphertext and its counter, e.g., \( Ctr \) in counter block \( C_0 \) (64B). We then...
need to update the main Merkle tree along the path from $C_0$ to $H_0$. Next, instead of updating the nodes along the path from $H_0$ to the root hash $R_0$, we update $H_0$ in the HNode cache and update the extra Merkle tree from $H_0$ to its root hash $R_2$. HNodeTree speeds up the integrity verification and update because the path on the extra Merkle tree (from $H_0$ to its root hash $R_2$) is much shorter than the path on the main Merkle tree (from $H_0$ to its root hash $R_0$).

**D. HNodeTree Management**

Since this extra MT is built on a newly added HNode cache, we next discuss how to manage this cache. We organize the HNode cache as a 4-way set associative cache and adopt a frequency based replacement policy. Out of the four entries in one cache set, we identify two as hot entries and the other two as warm entries (Figure 10(c)). We keep a frequency counter $F$ for each warm entry and an HMAC for each hot entry.

Each entry is an HMAC chosen from the main Merkle tree. Assume the main Merkle tree has $L$ levels, and the root and the leaf nodes (i.e., the counter blocks) are at level 0 and $L-1$, respectively. We adopt a simple strategy that chooses candidate nodes from level $L-2$. When an HMAC (8B) is generated from a leaf block, we treat the HMAC as a candidate entry and send the HMAC’s address $X$ to the HNode cache. In the computation, we reserve a special HMAC, i.e., all 1s for special use. If a normal block generates such an HMAC, we increment the corresponding counter in the leaf node to generate a different one.

The HNode cache adopts frequency based policy to identify the hot nodes — (i) if $X$ is a miss to the HNode cache, we replace the warm entry having the smallest $F$; (ii) if $X$ is a hit to one warm entry, we increment $F$; and if $F$ is now bigger than a threshold $T$, we replace the oldest the hot entry. Replaced hot entries become warm entries. (iii) if $X$ is a hit to one hot entry, or $X$ just becomes hot as above, we access the extra Merkle tree, either to update the tree (when servicing a write operation) or to verify the integrity (when servicing a read operation).

**Maintaining the two Merkle trees.** The extra Merkle tree is built using the hot entries of the HNode cache. Regarding the three designs issues of CacheTree, we choose to (1) use the special HMAC to indicate the involved HMAC nodes; (2) recompute the HNode cache contents from in-NVM counter blocks; and (3) sort the multiple HMAC nodes that are mapped to one cache size.

We need to update the extra Merkle tree when there is a change to the hot entries. There are two possibilities: case#1: we update the HMAC and it is a hit to the hot entry in the cache; or case#2: one hot entry is replaced with a new one. For both cases, we update the extra Merkle tree and update the root hash $R_2$ in the secure on-chip register.

For (case#2), we also need to update the main Merkle tree, i.e., extract the new hot entry from the main Merkle tree, and place the replaced hot entry back to the main Merkle tree. To extract the new hot entry, we persist a special HMAC, i.e., an 8B block with all 1s, to the NVM (the main memory) using its corresponding address in the main Merkle tree, shown in Figure 10(b). If a datablock generates the all 1s’ HMAC, we increment the leaf counter and update the corresponding MT nodes. The replaced hot entry is then sent back to the main Merkle tree, to ensure system recovery, we persist its HMAC to the NVM, i.e., overwriting the previously persisted special HMAC in NVM. For the main Merkle tree, we place the special HMAC at the corresponding place for the new hot node, and place the real HMAC at the corresponding place for the replaced hot node. At last, we update the extra Merkle tree using the new node to compute the new root hash $R_2$, and update the main Merkle tree to compute the new root hash $R_0$.

**E. System Recovery**

We next elaborate how the system may recover from a system crash or power failure. The first five steps are the same as in Section IV-D. They are to recover the counter, the counter cache, the MAC, the MAC cache, and the MACTree.

**Step 6**: We search the HMACs of the main Merkle tree in NVM. An HMAC being all 1s indicates the corresponding node has been moved to the HNode cache before crash. We thus compute its HMAC from its associated counter block and place it in the hot entries in HNode cache.

The controller sorts the hot entries in each cache set and then continuously build the extra Merkle tree till the root hash $R_2$. We compare $R_2$ with the one saved in the on-chip secure register and, if there is a mismatch, alarm the attack.

**Step 7**: We then reconstruct the main Merkle tree. We compute the hashes from the counters, replace the HMACs of all hot nodes to 1s, and iteratively construct the main Merkle tree till its root hash $R_0$. We then compare the computed $R_0$ with the one in secure register and, if there is a mismatch, alarm the attack.

**F. Security Analysis**

For BMT MT, the main MT nodes are saved in a reserved NVM region. These nodes are to speed up the integrity verification in the baseline. If lost, they can be reconstructed from the persisted counters. In HNodeTree, we use the special
HMAC, i.e., a block of being all 1s, to indicate that the corresponding main MT node has been moved to the HNode cache. Such an HMAC is persisted before the node being used to compute $R_2$ (before crash). An attacker may attack either the persisted special HMACs, or the other main MT node.

Since we avoid using this special HMAC in the main MT, any change to the location or the number of the special HMACs would result in constructing a different extra MT, which produces a different $R_2$ and thus leads to the detection of the attacks. An attack to other main MT nodes has no impact — the system reconstructs these nodes from the counters. If a computed one is different from the stored one, the system does not need to differentiate if it is because of an attack or losing the most up-to-date data.

### G. Comparison to Recent Art

**Difference with Anubis.** The most related work to ours is Anubis [44], which shadows the metadata caches in NVM and also builds a small tree for the shadow table to keep integrity. For each memory write, it records the updated metadata in shadow table (ST). The shadow table helps to recover the metadata rapidly after system crash. However, since ST resides in NVM, updating ST incurs additional NVM writes so that Anubis introduces performance and lifetime overhead.

CacheTree differs from Anubis in two aspects. Firstly, Anubis builds a small tree for ST that is in NVM. Anubis is a memory-based tree such that a memory write operation updates MAC in both metacache and ST (NVM). However, CacheTree builds extra MTs on cache contents, which eliminate NVM writes at memory write time. Secondly, their recovery schemes are different. Anubis, e.g., AGIT scheme, records explicit addresses of metadata in ST such that its metadata is kept in NVM-based ST when system crashes. It can locate dirty metadata directly during recovery. However, CacheTree loses all information in metadata cache when system crashes. For CacheTree to work, we need to satisfy the three design issues (as elaborated in Section III). As an example, MACTree uses the mismatch of ciphertext and MAC to identify the memory lines in the MAC cache; and HNodeTree uses the special HMAC. Multiple entries in cache set are sorted.

In conclusion, Anubis reduces the recovery time but introduces more NVM writes over the baseline. CacheTree builds small MTs on metadata caches such that it reduces NVM writes over the baseline. We will compare the recovery time of both schemes in the experiments.

**Difference with DMMT.** Another related scheme is dynamic multi-root merkle tree (DMMT) [23]. DMMT reduces integrity verification overhead by choosing a hot HMAC and store it in secure on-chip register, as a sub-tree root. The integrity verification and update can stop at this sub-tree root, hence improving the performance. HNodeTree addresses the two limitations in DMMT: (1) DMMT chooses one hot HMAC at a time. The nodes close to the root node tend to become hot early. However, choosing one such node leads to limited benefit. Choosing a node close to the leaf covers only a small memory region. Choosing more hot HMACs leads to large on-chip register overhead. HNodeTree dynamically covers a large number of nodes while using only one extra secure on-chip register. (2) The hot HMAC selection is expensive, which tends to stay unchanged for a long time. HNodeTree adopts a low cost hot node selection mechanism, which achieves good tradeoff between flexibility and performance improvement.

### VI. Methodology

To evaluate the effectiveness of CacheTree, we conduct experiments to compare it to the stat-of-the-art using a trace-driven in-house simulator. We use the PIN tool [2] to collect traces of SPEC CPU2006 [14] (ast, bwa, bzi, cal, gcc, hmm, lbm), CPU2017 [7] (lee r, mcfr) and the Persistence workload WHISPER [21] (echo). We utilize CACTI [20] to evaluate the added security caches. We run all the benchmarks for 400M instructions after skipping the warming up phase.

**TABLE I: Architectural configurations [35]**

<table>
<thead>
<tr>
<th>Processor</th>
<th>CPU</th>
<th>4 cores single issue in-order CMP, 2GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 I/D-cache</td>
<td>Private, 32KB, 2-way, 64B block, 2 cycles</td>
<td></td>
</tr>
<tr>
<td>L2 cache</td>
<td>Shared, 6MB, 8-way, 64B block, 20 cycles</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DDR-based PCM Main Memory</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Main memory</td>
<td>64GB, 4 channels, 2 ranks/channel, 8 banks/rank, 32-entry queue/channel</td>
</tr>
<tr>
<td>PCM latency</td>
<td>Read: 100ns, Write: 200ns</td>
</tr>
<tr>
<td>Energy</td>
<td>Read: 1.49nJ, Write: 6.76nJ</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Security Parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Counter Cache</td>
<td>Shared, 128KB, 4-way, 64B block 2 cycles</td>
</tr>
<tr>
<td>MAC Cache</td>
<td>Shared, 32KB, 4-way, 64B block, 2 cycles</td>
</tr>
<tr>
<td>MT Cache</td>
<td>Shared, 16KB, 4-way, 64B block, 2 cycles</td>
</tr>
<tr>
<td>HNode Cache</td>
<td>Shared, 8KB, 4-way, 64B block, 2 cycles</td>
</tr>
<tr>
<td>CacheTree Buffer</td>
<td>Shared, 6KB, 2 cycles</td>
</tr>
<tr>
<td>BMT</td>
<td>3 levels, 8-ary, 64B block in each level</td>
</tr>
<tr>
<td>MACTree</td>
<td>3 levels, 8-ary, 64B block in each level</td>
</tr>
<tr>
<td>HNodeTree</td>
<td>3 levels, 8-ary, 64B block in each level</td>
</tr>
</tbody>
</table>

Table I shows the details of the settings. Similar to prior work [5], [39], we assume the AES encryption latency to be 24 cycles. In pipelined process, the AES engine completes one encryption on each cycle [39]. We use a 6KB CacheTree buffer for MACTree and an 8KB HNode cache for HNodeTree. HNodeTree selects candidate HMAC nodes at level 7, i.e., parent nodes of leaves, and the hotness threshold is 20. We also study the sensitivity of these parameters in our experiments. We adopt the performance metric speedup as:

$$\text{Speedup} = \frac{\text{CPI}_{\text{base}}}{\text{CPI}_{\text{tech}}}$$

from [35] and [22], where $\text{CPI}_{\text{base}}$ and $\text{CPI}_{\text{tech}}$ are execution cycles of baseline and different schemes. In this paper, we mainly compare the following six schemes:

- **Baseline.** This is the baseline secure NVM design. It adopts BMT and Osiris [39]. For Osiris, a counter value needs to be persisted after 4 updates in the counter cache.
- **Synergy.** This scheme is built on the baseline. It adopts Synergy [28] to reduce MAC access overhead. Synergy has to use an extra parity.
- **Assure.** This scheme enhances the baseline using Assure [23] to reduce the verification overhead.
- **MACTree.** This scheme enhances the baseline using our proposed MACTree.
- **HNodeTree.** This scheme enhances the baseline using our proposed HNodeTree.
—CacheTree. This scheme combines MACTree and HNodeTree to reduce the integrity verification overhead.

VII. EVALUATION

A. Performance

We first compare the performance of different schemes and summarize the results in Figure 11. The numbers are normalized to the baseline. Compared to baseline, on average, Synergy and Assure reduce the execution time by 5.1% and 6.5%, respectively. As a pair comparison, MACTree outperforms Synergy by up to 11.9% (cal) while HNodeTree outperforms Assure by up to 13.4 % (ast). This is because, while Synergy improves the performance of MAC reads, it suffers from write-through parity update to NVM at MAC update time. The hot subtree root selected in Assure tends to be close to the root, which leads to limited benefits.

By enabling the write-back policy for the MAC cache, MACTree reduces the number of MAC writes significantly and achieves large system performance improvements. HNodeTree selects hundreds of hot HMACs into the HNode cache, which greatly reduces the number of MT nodes to be verified and updated. By combining MACTree and HNodeTree, CacheTree achieves on average 26.2%, 20.1% and 18.5% performance improvements over baseline, Synergy and Assure, respectively. In summary, CacheTree effectively mitigates the integrity verification and update overhead for secure NVMs.

B. Lifetime

We next study the NVM writes of different schemes and summarize the results in Figure 12. In the study, we add an ideal scheme WriteBack, which adopts the write-back policy for the MAC cache. As a reference scheme to evaluate the effectiveness of CacheTree, WriteBack does not persist any MAC data to help system recovery from system crashes.

Baseline and Synergy have the same numbers of NVM writes because Synergy does not reduces the numbers of NVM writes. Compared to baseline, Assure and HNodeTree reduce the numbers slightly, i.e., 0.8% and 1.5%, respectively, by accessing short paths in MT.

By adopting the write-back policy, WriteBack achieves the large reduction, i.e., 44.2%, over the baseline. MACTree needs slightly more NVM writes than WriteBack due to metadata update. CacheTree reduces on average 44.3% NVM writes over the baseline. CacheTree sometimes is slightly better than WriteBack as it combines the benefits from both MACTree and HNodeTree. In summary, CacheTree is an effective mechanism to prolong NVM chip lifetime for secure NVMs.

C. Energy consumption

We next evaluate the energy consumption of different schemes. For simplicity, we focus on read and write energy consumption of NVM. Figure 13 presents the results, with the numbers normalized to the baseline. From the figure, we observe that write energy consumption occupies the major portion. This is because many reads hit the cache while many writes are sent to NVM to ensure data persistence.

Compared to baseline, Synergy reduces 65.1% of read energy consumption by avoiding reading MACs. However, it only reduces the total energy consumption by 0.4%. Assure and HNodeTree are slightly worse than Synergy, which consume 0.2% and 0.3% less total energy, respectively, than the baseline. Due to generating fewer NVM writes, CacheTree consumes 44.0%, 43.7% and 43.9% less energy than baseline, Synergy and Assure, respectively. In summary, CacheTree significantly reduces the energy consumption for secure NVMs.

D. Sensitivity to MAC Cache Size

We next study the sensitivity of CacheTree to different parameters and summarize the results in Figure 14. Since the energy consumption correlates mainly to the NVM writes, the figure only reports the performance and lifetime results.

Figure 14(a) summarizes the results of the sensitivity study on MAC cache size. The results are normalized to the setting with the smallest MAC cache size (256B). From the figure, a small cache tends to evict dirty MAC cacheline more frequently and thus introduces more NVM writes; a large cache consolidates more writes but incurs large storage overhead. With increasing cache sizes, the number of NVM writes decreases while the performance improves. Due to a jump of the number of MACTree levels at 8KB and 64KB, there are two performance improvements dips. In this paper, we choose the MAC cache size to be 32KB as it achieves a good tradeoff among performance improvement, NVM writes, and storage overhead.

E. Sensitivity to HNode Cache Size

Figure 14(b) evaluates the sensitivity of the scheme to the HNode cache size, with the results normalized to that of a 32B HNode cache. A small cache can only keep a limited number of hot HMACs and thus results in limited performance improvement. A large cache leads to a large extra Merkle tree and thus incurs large storage overhead.

From the figure, increasing HNode cache size leads to a small fluctuation in the number of NVM writes while it achieves steady performance improvement. The performance improvement peaks at 8KB size, which needs a full 8-ary 3-level extra Merkle tree. A larger cache results in more levels of the extra Merkle tree, which leads to a slight performance decrease. In this paper, we choose an 8KB HNode cache.

F. Sensitivity to Candidate Hot Node Levels

In HNodeTree, we choose nodes at a fixed level $l$ from the main Merkle tree and send them to the extra Merkle tree. The selection of $l$ also impacts the system performance. Figure 14(c) reports the results with different $l$ values.

From the figure, we observe that the level value has negligible impact on NVM writes. When selecting HMACs from the bottom levels of the main Merkle tree (i.e., towards to the leaf nodes that have large $ls$), we tend to achieve
large improvements due to skipping more nodes on the main Merkle tree. However, such a selection has low locality, which demands selecting more nodes to cover a larger user space. Choosing the HMACs close to the root hash has better locality but tend to have little performance improvement (as we have already gone through the long verification path). In this paper, we choose hot HMAC candidates at level 7 with the highest performance, which is just next to the leaf nodes.

G. Sensitivity to Hotness Threshold

At last, we study the sensitivity of the scheme to the hotness threshold. Figure 14(d) summarizes the results when we range from 2 to 64.

From the figure, choosing a small hotness threshold tends to introduce a large number of hot nodes, which flushes the hot entries in the HNode cache frequently. Given we need to update the main MT when there is a change of the hot nodes, the performance improvement is low when choosing a small hotness threshold.

We also observe a performance jump with increasing hotness threshold values and then a gradual decrease when the values are bigger than 20. This is because not many hot nodes can be selected if the threshold value is too big, which reduces the opportunities to reduce the integrity verification and update overhead. In this paper, we set the threshold to 20, which achieves a good tradeoff between performance and lifetime.

H. Overhead

CacheTree adds small storage overhead, including an 8KB HNode cache, a 6KB CacheTree buffer to buffer the internal nodes of the extra Merkle trees, and two non-volatile secure registers, which is less than 0.5% of all cache capacity. CacheTree also introduces additional computation and access overhead to the extra Merkle trees. Due to the adoption of AES pipeline engine, the HMAC computation overhead is insignificant. Since we save all the nodes of two extra Merkle trees in on-chip buffers, the accesses can overlap with other cryptographic operations, which result in low access overhead.

I. Comparison with SGX-style MT and Other Solutions

We compare CacheTree with the state-of-the-art SGX-style MT designs, including the original SGX Merkle tree (SGX), VAULT [34], and Morphable counters (MCtr.) [27]. For 64GB NVM, they use 10-level, 7-level and 4-level Merkle trees, respectively. AGIT and ASIT are two Anubis schemes for BMT and SGX-style Merkle trees, respectively. Based on Baseline, a NV cache solution (NVcache) that utilizes non-volatile memories e.g., STT-RAM [41], to construct MAC.
and MT caches, so that persists updates in caches. Similar to existing works [38], [45], the last solution utilizes the 32-SRAM-entry Write-Pending-Queue (WPQ32) and an enhanced 64-SRAM-entry one (WPQ64) to persist updates and coalesce related NVM writes due to locality principle [29]. The number of entries of WPQ is limited due to its limited power source.

Figure 15 and Figure 16 summarize the performance and NVM writes of different schemes, normalized to the baseline. Compared to baseline, SGX, VAULT and MCtr have on average 61.3%, 50.4% and 28.1% performance degradation, respectively; and 3.28×, 2.13× and 1.06× more NVM writes, respectively. Generally, VAULT has less NVM writes and achieves better performance than SGX for most workloads. This is because VAULT has fewer tree levels that need to be persistent. Similarly, MCtr is better than VAULT due to fewest tree levels. For Anubis, AGIT and ASIT have more NVM writes than Baseline due to shadowing overhead. Compared to Baseline, NVcache increases the performance by 14.2% averagely while leads to 2.89× more NVM writes. Here, the NVM writes of NVcache contain writes of NV caches and NVM. It absorbs many original NVM writes in NV caches and MT updates results in writes in NV caches. WPQ32 and WPQ64 achieve both of performance and lifetime improvement than Baseline since they coalesce many NVM writes in memory controller.

For performance, CacheTree outperforms SGX, VAULT and MCtr by 2.27×, 1.54× and 0.76×, respectively. It is 0.39× and 0.21× better than AGIT and ASIT, respectively. And it increases performance by 10.4%, 14.6% and 12.9% than NVcache, WPQ32 and WPQ64, respectively. All of NVcache, WPQ32 and WPQ64 suffer from performance overhead of MT updates.

CacheTree introduces the least amount of NVM writes of all schemes, e.g., it achieves 48.9%, 52.5%, 12.7% and 4.3% NVM writes reduction from AGIT, ASIT, WPQ32 and WPQ64, respectively. Note that WPQ64 absorbs more NVM writes than WPQ32, closer to lifetime of CacheTree. There

J. Recovery Time Comparison

We then compare the recovery time of different schemes in Figure 17. From the figure, Baseline and CacheTree need increasingly longer recovery time with increasing NVM capacity. Since CacheTree needs to recover extra MAC-Tree and HNodeTree, it costs about 23.8% more time than Baseline. However, for GB-level NVM system, e.g., 64GB in this paper, CacheTree only takes about 139s to recover, which is acceptable. AGIT and ASIT need around 0.05s recovery time, and the time depends on the ST capacity [44].

For larger capacity NVM systems, e.g., TB-level NVM, Baseline and CacheTree need much longer time to recover. In this scenario, it is preferable to combine CacheTree with AGIT to reduce recovery time. That is, once CacheTree locates dirty MACs and hot HMACs rapidly, CacheTree can realize a similar recovery time as that of AGIT. Consequently, CacheTree+AGIT adopts a shadow table (ST) in NVM to record addresses of dirty counter blocks, MT blocks and MAC blocks as well as hot HMACs. The ST capacity (in NVM) is 184KB (128KB for counter, 16KB for MT, 32KB for MAC and 8KB for hot HMAC). Note that the combined scheme records the addresses of dirty MAC blocks instead of MAC contents. Due to access locality, this ST incurs limited NVM writes. During recovery, the shadow table helps to locate dirty counters, dirty MT nodes, dirty MACs and hot HMACs. Thus, CacheTree+AGIT can recover the counters, MT, MACTree and HNodeTree rapidly. And the on-chip roots R₀, R₁ and R₂ can ensure the integrity of recovered data. As shown in this figure, CacheTree+AGIT only needs about 0.08s to recover regardless of the NVM capacity.

We then evaluate the overall effectiveness of the combined scheme. Since ST introduces extra NVM writes, CacheTree+AGIT incurs 8.9% performance and 15.4% lifetime benefit losses over CacheTree. It still achieves 27.3% performance and 40.8% lifetime improvements over AGIT.

K. Area Overhead

In this section, we evaluate the area overhead of all solutions, as shown in Table II. Since some solutions introduce
extra on-chip components, we evaluate the area overhead at the cache level. We utilize NVSim [10] to compute the area overheads of the NV cache solution. Some solutions, i.e., Synergy, SGX, VAULT, and WPQ, do not incur area overhead as they add no additional components. Anubis needs an additional Merkle tree for shadow table, leading to 0.47% of area overhead. NV cache replaces SRAM cache with STT-RAM, achieving a smaller cache due to high density of STT-RAM. However, it suffers from high manufacturing cost and large runtime energy consumption and performance overhead. CacheTree adds two small additional merkle trees and a HNode cache, resulting in 0.34% of area overhead. In conclusion, CacheTree has negligible area overhead.

### TABLE II: Area overhead of different solutions

<table>
<thead>
<tr>
<th>Solutions</th>
<th>Area overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synergy</td>
<td>0</td>
</tr>
<tr>
<td>Assure</td>
<td>0.19%</td>
</tr>
<tr>
<td>SGX</td>
<td>0</td>
</tr>
<tr>
<td>VAULT</td>
<td>0</td>
</tr>
<tr>
<td>AGIT</td>
<td>0.47%</td>
</tr>
<tr>
<td>ASIT</td>
<td>0.47%</td>
</tr>
<tr>
<td>NVcache</td>
<td>-84.8%</td>
</tr>
<tr>
<td>WPQ</td>
<td>0</td>
</tr>
<tr>
<td>CacheTree</td>
<td>0.34%</td>
</tr>
</tbody>
</table>

## VIII. RELATED WORK

In this section, we briefly summarize the related studies other than those discussed in preceding sections. Swami et al. proposed to integrate smart encryption with XOR-based energy masking to realize low write energy/latency and improve the lifetime in secure MLC and TLC NVMs [32]. Young et al. proposed to re-encrypt only modified words when servicing a memory write [40]. Chhabra et al. proposed i-NVMM to lower encryption/decryption overhead by keeping hot cachelines in unencrypted form in the memory [8]. Swami et al. proposed on-demand memory allocation to mitigate the memory encryption frequency with negligible memory overhead [31]. Liu et al. enforced selective counter atomicity and relaxes persisting counters for non-persistent data [17]. Liu et al. proposed Janus to exploit memory parallelism by memory operation decomposition and start sub-operations once the inputs are ready [18]. It is orthogonal to the CacheTree design.

The proposed CacheTree differs from existing schemes as it enables the write-back policy for the MAC cache and helps to reduce the integrity verification and update overhead. In addition, CacheTree can combine with existing schemes, e.g., with Osiris [39] to reduce counter persisting overhead, and with AGIT [44] to reduce the system recovery time.

## IX. CONCLUSION

In this paper, we proposed CacheTree to address the challenges in developing a high performance secure NVM design. By creating extra Merkle trees on security metacache contents, CacheTree enables the adoption of write-back policy and thus greatly reduces the number of NVM writes. Our experimental results show that CacheTree, with less than 0.5% storage overhead, achieves up to 20.1% performance improvement, 44.3% lifetime increase and 43.7% energy consumption reduction over the state-of-the-art solutions.

## ACKNOWLEDGMENT

This work is supported by the National Key Research and Development Program of China under Grant NO. 2019YFB1804502, the National Natural Science Foundation of China under Grant NO. 61832020, 61802418, the Natural Science Foundation of Guangdong Province under Grant No. 2018B030312002, the Major Program of Guangdong Basic and Applied Research under Grant No. 2019B030302002, the Program for Guangdong Introducing Innovative and Entrepreneurial Teams under Grant NO. 2016ZT06D211, Key-Area Research and Development Program of Guangdong Province under Grant No. 2019B010107001. The authors thank the anonymous reviewers for their constructive comments.

## REFERENCES
