No Compromises: Secure NVM with Crash Consistency, Write-Efficiency and High-Performance

Fan Yang, Youyou Lu*, Youmin Chen, Haiyu Mao, Jiwu Shu*
Department of Computer Science and Technology, Tsinghua University, Beijing, China
Beijing National Research Center for Information Science and Technology (BNRist)
{yangf17@mails.,luyouyou@,chenym16@mails.,mhy15@mails.,shujw@}tsinghua.edu.cn

ABSTRACT

Data encryption and authentication are essential for secure NVM. However, the introduced security metadata needs to be atomically written back to NVM along with data, so as to provide crash consistency, which unfortunately incurs high overhead. To support fine-grained data protection without compromising the performance, we propose cc-NVM. It firstly proposes an epoch-based mechanism to aggressively cache the security metadata in CPU cache while retaining the consistency of them in NVM. *Deferred spreading* is also introduced to reduce the calculating overhead for data authentication. Leveraging the hidden ability of data HMACs, we can always recover the consistent but old security metadata to its newest version. Compared to Osiris, a state-of-the-art secure NVM, cc-NVM improves performance by 20.4% on average. When the system crashes, instead of dropping all the data due to malicious attacks, cc-NVM is able to detect and locate the exact tampered data while only incurring extra write traffic by 29.6% on average.

1 INTRODUCTION

The emerging byte-addressable non-volatile memory (NVM) technologies, such as PCM, STT-RAM, ReRAM and Intel 3D XPoint, have the advantage of high density, low-power and high scalability, making them a promising alternative to DRAM as the main memory. Further, these new memory technologies bring data durability to the in-memory system, which blurs the difference between storage and memory, making it possible to store and manipulate persistent data in-place in memory [1, 5, 7, 8].

Since NVMs are directly attached to memory bus (a.k.a., persistent memory, PM), they are vulnerable to malicious attacks similar to that DRAM system may suffer from. There are mainly two types of attack models: ① data confidentiality attack (i.e., stealing the privacy data), and ② data integrity attack (e.g., data spoofing, splicing, and replaying). In response to the above two attack models, existing secure memory system introduces counter mode encryption (CME) [11, 17, 19, 22, 24, 25] and Merkle Tree (MT) authentication [3, 18, 19, 23] to respectively ensure data confidentiality and integrity. In CME, each time a data block is evicted out of the CPU cache, it is encrypted with a unique counter and the counter then increases by one. Bonsai MT (BMT) is the state-of-the-art MT authentication architecture. To prevent spoofing and splicing attacks, BMT places a single layer of hash message authentication codes (data HMACs) alongside the data. It also builds a hierarchical tree structure of counter HMACs over counters to detect replay attacks.

To improve performance, traditional DRAM-based secure memory systems store these frequently accessed security-support metadata (aforementioned counters, HMACs) in dedicated cache space [4, 6, 17] or directly store them in the last-level cache [3, 19]. However, metadata caching is non-trivial for secure NVMs. The cached data in CPU cache may lose after a system/power failure, thus leading to the corrupted state when the data reaches NVM while the relevant metadata doesn't. Consequently, data decryption and authentication may fail when the system is rebooted. As a result, how to achieve crash consistency [14, 15] becomes the building block for secure NVM systems. Specifically, it's important to guarantee ❶ the consistency between data and secure metadata and ❷ the internal consistency of the Merkle Tree simultaneously. However, implementing such a crash consistency mechanism is expensive. We have implemented a naive approach, which aggressively evicts the cached secure metadata. It can significantly deteriorate the system performance by 41.4% and increase memory write traffic by 5.5×, in comparison to the NVM-based secure system without consistency guarantee. The root cause of such inefficiency lies in two aspects: (1) *Write amplification.* Each time a data block is evicted, all the related encryption counters and the tree nodes in the Merkle Tree need to be flushed to NVM as well. (2) *Cascading calculating on the Merkle Tree.* Once a counter is incremented, all its ancestors till the root node in Merkle Tree need be recalculated.

In this paper, we propose a secure NVM system named cc-NVM. It is capable of *detecting and locating the attacks in NVM both at runtime and after system crashes, while introducing minimal crash consistency overhead.* By exploring the hidden ability of data HMAC, we are enabled to lazily flush encryption counters to NVMs. In the absence of spoofing/splicing attacks, any stalled counters after a system failure can be safely recovered. Based on this principle, we then propose epoch-based consistent Merkle Tree. It aggressively caches the tree nodes in CPU cache and atomically synchronizes the updates to NVM. As a result, the Merkle Tree in NVM is consistently transited from an old state to new state. Leveraging the consistency property of the Merkle Tree in NVM, the ability to

*Jiwu Shu and Youyou Lu are the corresponding authors. This work is supported by National Key Research & Development Program of China (Grant No. 2018YFB1003301), the National Natural Science Foundation of China (Grant No. 61832011, 61772300), Research and Development Plan in Key field of Guangdong Province (Grant No. 2018B010109002), and Huawei Innovation Research Program (Grant No. 2017070004).

DAC '19, June 2–6, 2019, Las Vegas, NV, USA

ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317869

thwart replay attacks still holds. To reduce the HMAC calculating overhead, similar to existing DRAM-based secure system, cc-NVM calculates the new tree nodes from the bottom up, and stops if it has already been cached. Instead, the root node is lazily updated only at synchronization. Our evaluations show that cc-NVM improves IPC by 20.4% and only incurs extra write traffic by 29.6% when compared to Osiris [9], but with the ability to detect/locate attacks after crashes.

Optimizations on secure NVMs have been studied recently [9, 10, 12, 17]. Comparatively, cc-NVM is novel in the following ways:

- Since the Merkle Tree in NVM is guaranteed to be always consistent, we can not only detect attacks but also locate the exact tampered data block even after a system failure.
- With an epoch mechanism, cc-NVM fully exploits the benefits of metadata caching with minimal crash consistency overhead.

2 BACKGROUND AND MOTIVATION

2.1 Threat model

Our thread model identifies two regions of a system just as in prior studies on hardware-based memory encryption and authentication. The secure region named trusted computing base (TCB) consists of the processor chip and core parts of the operating system (e.g., security kernels). Any on-chip code or data (i.e. in registers or caches) is considered safe and cannot be observed or manipulated by attackers. The non-secure region includes all off-chip resources, primarily including the off-chip memory and the processor-memory bus [13, 19, 21]. The attacker can obtain secret values stored in memory or transferred through off-chip interconnects when the program is running, or directly steals the NVM DIMM (i.e., *data confidentiality attacks*). The adversary can also act as a man-in-the-middle to modify values in the physical memory and on all off-chip interconnects. This is called *data integrity attacks*, which include tampering the value directly (spoofing), exchanging the content of one address with another's from a different location (splicing), and replaying an old value at the same location (replay).

2.2 Encryption and authentication in NVMs

Memory encryption and authentication techniques are widely used to detect confidentiality attacks and integrity attacks.

Memory encryption is to ensure that the adversary cannot obtain any meaningful data stored outside the TCB. Therefore, any data evicted out of the CPU cache should be encrypted before being stored in memory. Counter mode encryption (CME) is the most frequently used approach to encrypt memory data. CME encrypts a data block by XORing it with a one-time pad (OTP) [21]. Specifically, OTP is generated with a secret key and a seed as inputs (the seed consists of the address of the data block and an associated counter). Conversely, when a data block is fetched from memory, the same seed is encrypted to generate the same OTP, and then we use it to decrypt the data block. CME is ensured to be secure by guaranteeing the uniqueness of each seed: (1) different data blocks are mapped to different counters; (2) the counter is increased by 1 for each data write-back (encryption).

Memory authentication is devoted to guaranteeing the integrity of in-memory data. The state-of-the-art memory authentication architecture is the Bonsai Merkle Tree (BMT) [18, 19] (shown in Figure 1), which is deployed on the CME architecture. It first uses a single layer of message authentication codes (a.k.a, data HMACs) to detect spoofing and splicing attacks, where each HMAC is a hashed value generated by taking the encrypted data block, counter and its address as inputs. It then builds the Merkle Tree (MT), a hierarchical tree structure, to prevent replay attacks. Each leaf node of the Merkle Tree is a memory line of counters, and the parent nodes (a.k.a, counter HMACs) store the HMACs of its children computed using a keyed hashing function (e.g., HMAC based on SHA-1). The secret HMAC key and the root of MT are stored in secure on-chip registers to prevent the MT from being replayed from bottom up. BMT is different from the traditional approaches which build MT over both counters and data [3]. In BMT, the encrypted data blocks are not directly protected. However, it is immune to replay attacks since the data HMACs include the MT-protected counters as input. Therefore, BMT has lower metadata storage overhead, thus shortening the tree depth and reducing the MT read/write times.

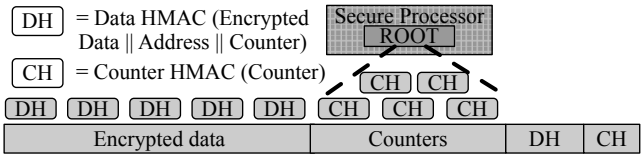

Figure 1: Data Layout of Bonsai Merkle Tree

Caching security metadata can boost performance for both CME and BMT. Take a memory read and decryption for example, if the corresponding counter (in CME) has already been cached, the OTP generation and the read access can be executed in parallel, hiding the OTP generation latency. Typically, those counters of different data blocks in the same data page (i.e., 4 KB) are organized into the same cache line [19, 24]. Therefore, almost all the workloads tend to have high cache hit ratio of metadata. The frequently accessed and verified tree nodes in Merkle Tree can be cached on-chip too [3]. This allows the integrity verification of a data block to complete as soon as the needed tree node is found in the on-chip cache. The reason being, since the cached tree nodes have already been verified and their security is guaranteed being on-chip.

2.3 Crash consistency cost in secure NVMs

While placing the frequently accessed secure metadata into on-chip cache improves performance, it causes consistency problems for NVMs. Once a system/power failure occurs, the cached metadata may lose, while its associated data might have already reached NVM. Such inconsistency between data and metadata can prevent us from correctly decrypting and authenticating the NVM data. To guarantee that we can always use the correct data at runtime and retain secure protection after crashes, we need to ensure that:

(1) The data block, the associated counter and data HMAC should reach memory atomically. Otherwise, the data cannot be decrypted or authenticated correctly.
(2) All layers of the Merkle Tree in NVM along with the root node in TCB should be consistently updated.

Specifically, updating of MT is extremely expensive: Updates should be applied from the leaf node till the root node (12 layers for a 16 GB NVM with 128-bit HMAC). What's more, the calculation of each HMAC in the tree nodes must be executed one after another (instead

of in parallel), since the parent node stores several counter HMACs, and each of them is generated with one of its child nodes. Thus, the write-back should wait until the root is updated.

These restrictions significantly limit the efficiency of secure NVM. We have implemented a naive approach (called SC in Section 5) to guarantee consistency, which ensures atomicity by aggressively flushing all the related security-support metadata. Our evaluation shows that it can increase memory writes by 5.5× and deteriorate system performance by 41.4%, when compared to conventional security architecture without crash consistency guarantees.

3 RELATED WORK

Memory encryption. [17] proposes counter-atomic mechanism by exploiting the application-level persistence semantics. It can selectively relax the atomicity between the data blocks and counters. However, it requires extensive changes to existing applications and can cause encryption pad reuse for non-persistent memory locations [9]. Osiris [9] introduces extra ECC bits to correctly recover the counters even when they are lost. Arsenal [12] ensures the atomicity between the data block and the related metadata by compressing the data block. Therefore, the data block, data HMAC and counter can be placed into a single cacheline.

Memory authentication. [9, 12, 23] propose that the Merkle Tree can be successfully reconstructed so long as the data block, the corresponding counter in the NVM (i.e., leaf node) and the root node in TCB are updated consistently. Therefore, the inner nodes of the Merkle Tree don't need to be flushed out of the CPU cache for each written-back data block. However, since the data block and the root node need to be updated atomically, the write-back data will be blocked until the root node is calculated. This approach ensures that the replay attacks can always be detected if the constructed root node mismatches with the root node stored in TCB (a non-volatile register). However, it is unable to point out the exact data block that has been corrupted: If a replay attack happens, it's possible that all the data HMACs match with their corresponding counters and data blocks, but the reconstructed root node mismatches with the TCB one. Actually, any spoofing and splicing attacks can prevent the detection of replay attacks, because when these two types of attacks happen, the reconstructed root from the remaining leaves can't match with the TCB root. As a result, all the data in NVM should be dropped once an integrity attack occurs.

In conclusion, existing approaches either are vulnerable to integrity attacks [10, 17], or cannot pick out the tampered data after a system failure [9, 12]. In addition, existing approaches [9, 12] cannot avoid the aforementioned high calculating overhead when updating the Merkle Tree.

4 DESIGN

4.1 Overview

Figure 2 shows the overall architecture of cc-NVM. To improve performance, both the encryption counters and tree nodes of the Merkle Tree (i.e., counter HMACs) are cached in *Meta Cache*. The key insight behind the cc-NVM design lies in *how to exploit the benefit of metadata caching, while providing fine-grained data protection even after system crashes*. For this purpose, cc-NVM incorporates three components working together in the memory controller, which are *Encryption Engine*, *Drainer* and *Write Pending Queue*.

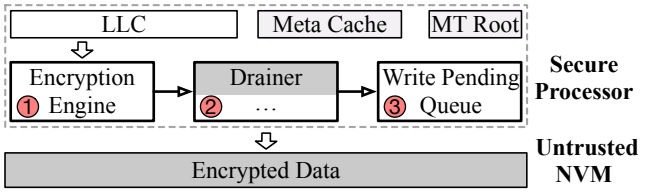

Figure 2: Architecture of cc-NVM.

It firstly uses an epoch-based mechanism to guarantee the consistency of BMT when exploiting the benefits of metadata caching (in Section 4.2). For normal write-back data blocks, the *Encryption Engine* updates the corresponding secure metadata directly in *Meta Cache*. Meanwhile, the addresses of all dirty cachelines in *Meta Cache* are recorded by the *Drainer*. Once a draining event is triggered, the *Drainer* then atomically commits the updates in the current epoch to the *Write Pending Queue* and finally to NVM. Therefore, the Merkle Tree in NVM is guaranteed to be always consistent, so its ability to specify the replay attacks still holds. Besides, by aggressively caching the metadata in *Meta Cache* and lazily evicting them out, write traffic to NVM is dramatically reduced.

To reduce the overhead of calculating the counter HMACs for each data write-back event, the *Encryption Engine* stops calculating the counter HMACs for a tree node once its child has already been cached in *Meta Cache*. Instead, the update is spread to the root node only at draining phase (in Section 4.3). Thus the counter HMACs computation overhead for each epoch is reduced.

After a system failure, however, a consistent but "old" Merkle Tree in NVM may mismatch with the newest data blocks and data HMACs. As a result, the subsequent process of data decryption and integrity checking may fail. Luckily, we observe that the data HMACs can be leveraged to recover those stalled counters to their newest versions (in Section 4.4). Following this, we can further rebuild the Merkle Tree with the newest counters and finally detect/locate all possible attacks.

4.2 Epoch-based Consistent BMT

To better utilize the *Meta Cache* while being immune to malicious attacks, we propose *epoch-based consistent BMT*. It achieves this by aggressively caching the frequently accessed secure metadata in *Meta Cache* and atomically committing these updates to NVM. The duration between two adjacent committing points is known to be an epoch.

Aggressive Caching. For normal write-back events, as shown in Figure 3, each time a data block is evicted from the last level cache (in ❶), it is encrypted and authenticated by the *encryption engine* (in ❷). In this step, all the metadata updates (e.g., adding 1 to the corresponding counter, modifying the related tree nodes in MT) are conducted directly in *Meta Cache*. The root node in TCB is updated too. In the meantime, the drainer tracks all the dirty cachelines in *Meta Cache* by appending the related addresses of them into its *dirty address queue* (in ❸). Note that we skip those dirty cachelines if their addresses have already been put in the *dirty address queue*. Actually, the process of ❷ and ❸ is executed in parallel, this is because for a specific data block, the related metadata addresses are deterministic in the existing secure NVM system.

While caching the metadata reduces write traffic to NVM, we still need to periodically commit the cached updates to NVM. There

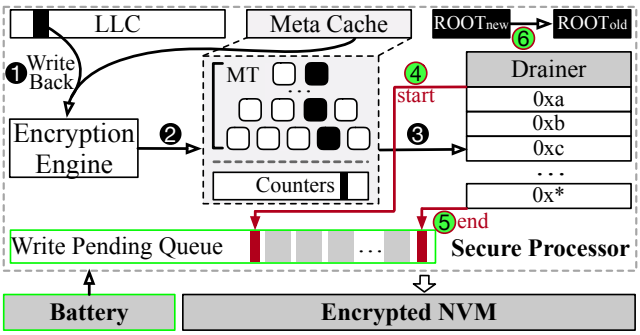

Figure 3: Epoch-based Consistent BMT.

are mainly two challenges: 1) How to guarantee the atomicity of committing since the metadata in NVM needs to be consistently updated; 2) How frequent should we commit (i.e., epoch length), which balances the gains of caching and the risk of data loss.

Atomic Draining. To address the first issue, we adopt the hardware Asynchronous DRAM Refresh (ADR) [20] mechanism. It ensures that any write request buffered in the write pending queue (WPQ) of the memory controller will be successfully written back to NVM with some backup power in case of a power failure. Thus, the number of entries in WPQ limits the size of the persistent domains in processors. In our implemented system, we use a 64-entry (i.e., 4kB) WPQ. We then add two persistent registers in TCB. among them, $ROOT_{new}$ is updated for normal write-back events in step ❷, while $ROOT_{old}$ is updated only at committing phase.

Based on this, we propose atomic draining protocol. To atomically commit the updates to NVM, the drainer firstly sends a *start* signal to the memory controller (step ❹), and flushes the dirty cachelines tracked in the *dirty address queue* to the memory controller. Once the memory controller receives the signal, it starts to block the metadata cachelines by putting them in the WPQ (normal data blocks still flow in legacy mode). When all the related metadata cachelines have been sent to the memory controller, the drainer sends an *end* signal to notify the controller to flush all the cachelines in WPQ to NVM (step ❺). Finally, the drainer update $ROOT_{old}$ with the value of $ROOT_{new}$, so $ROOT_{old}$ is consistent with the Merkle Tree in NVM (step ❻). When a system failure occurs, the memory controller continues to flush the cachelines in WPQ to NVM with the backup power provided by ADR subsystem. However, if the system crashes before the memory controller receives the *end* signal, it just drops all the residual cachelines in the queue, so as to keep the Merkle Tree in NVM consistent. In addition, when the system crashes after the memory controller receives the *end* signal while before updating the $ROOT_{old}$, we are sure that the newest consistent state of the Merkle Tree will eventually reach NVM because of the ADR support. So the Merkle Tree in NVM is consistent with the $ROOT_{new}$ and it still can detect/locate attacks. In conclusion, we can ensure that the Merkle Tree in NVM is always consistent with at least one of the root in TCB.

Epoch Length. To address the second issue, atomic draining is triggered when any one of the following events is satisfied:

(1) the *dirty address queue* is full or it doesn't have enough entries to store the corresponding metadata addresses of the next evicted data block; In our implementation, the *dirty address queue* and WPQ has the same number of entries.

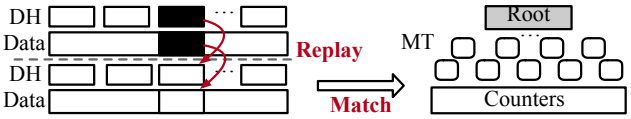

Figure 4: Undetectable Replay Attack with Old Root.

(2) a cacheline in *meta cache* is evicted by the cache system;

(3) a cacheline in *meta cache* has been updated for more than N times since it becomes dirty (for fast recovery, in Section 4.4).

Besides, when a draining event is triggered, step ❶ and ❷ for the subsequent evicted data blocks is blocked until the draining is finished. With such epoch-based consistency mechanism, applications can fully exploit the benefits of metadata caching by maximizing the epoch length. Meanwhile, the Merkle Tree in NVM is guaranteed to be consistently transited from an old state to a new one.

4.3 Deferred Spreading

Considering the locality property of most workloads, it's very likely that several adjacent data blocks evicted by CPU cache share the same ancestors of tree nodes in MT. Therefore, updating the tree nodes for each data block independently can cause great redundancy of computing (in step ❷). To eliminate such redundancy, we propose *deferred spreading* by calculating and updating the old tree nodes of Merkle Tree only at draining phase.

Different from that depicted in Section 4.2, in step ❷, we stop updating the counter HMAC when it has already been cached in *Meta Cache*. This is because the verified and cached tree nodes are always considered safe, which shares the same principles as that in traditional DRAM-based secure system [3]. In step ❸, we still need to reserve entries in the *dirty address queue* for the related metadata counters and counter HMACs, despite the fact they have not been dirtied yet. During the draining phase, all the tree nodes (i.e., counter HMACs) indexed by the *dirty address queue* are finally calculated and updated by the encryption engine. $ROOT_{new}$ in TCB is updated accordingly. In this way, the same tree node only needs to be calculated once for each draining.

Potential Replay attacks. However, introducing *deferred spreading* can lead to undetectable replay attacks: As shown in Figure 4, a system failure occurs before the last draining is committed. So the Merkle Tree along with the root node is still in an old state, despite that several data blocks have been newly written. If the newly written data and the associated data HMAC are replayed to their old version, such replay attack becomes undetectable because (1) the old Merkle Tree in NVM is consistent with one of the roots in TCB (both of them are lazily updated with the optimization in Section 4.3); and (2) the old counter still matches with the replayed data block and data HMAC. Actually, the key reason for such potential replay attack is that the consistency between the data blocks and Merkle Tree is relaxed. By lazily updating the $ROOT_{new}$, the newly written data blocks is not always protected by an old Merkle Tree. To avoid such undetectable replay attacks, we add an extra 64-bit persistent register (a.k.a, N_{wb}) in TCB, so as to record the number of write-back events since the last draining is committed. During recovery, we then use this register to detect such replay attacks (in Section 4.4).

4.4 Crash Recovery and Attack Locating

When the system crashes before a draining is committed, the Merkle Tree in NVM may mismatch with the data blocks and data HMACs.

So we need a way to recover it to its newest version. We observe that the data HMACs can be leveraged for crash recovery: During runtime, a counter is incremented by one each time a data block is written-back, so we can always recover a stalled counter to its newest version by calculating the data HMAC with this counter and the corresponding data block, and comparing it with the existing data HMAC in NVM. If they are equal, this counter is believed to be the newest. Otherwise, we increase the counter by one and retry. To correctly work with such recovery mechanism, the data block and data HMAC should be flushed to NVM atomically for each write-back event. Luckily, this is not an issue since in existing Bonsai MT architecture, data HMACs are not cached in *meta cache* [19], and are generated directly in the memory controller. Therefore, such atomicity can be easily achieved with the existence of ADR.

Crash Recovery. Since counters can be correctly recovered, the Merkle Tree can be reconstructed too. Three types of integrity attacks are detected and located during the recovery process. We divide the recovery and attack detection process into four steps:

1) *Locate normal replay attacks.* With the epoch-based consistency mechanism, the MT in NVM is ensured to be always consistent with at least one of the roots in TCB in the absence of any attacks. Therefore, a replay attack can be located if any two of parent and child tree nodes mismatch.
2) *Recover the stall counters and locate attacks.* When recovering the counters with the above method, if the generated data HMAC fails to match after N times of retrying (N is a trigger event described in Section 4.2), we can judge that either the data block or the data HMAC has been attacked (i.e., spoofing/spicing).
3) *Detect potential attacks.* When recovering the counters, we record the total number of retries (N_{retry}). If N_{wb} (in Section 4.3) and N_{retry} are not equal, we assert that an attack occurs.
4) Rebuild the Merkle Tree based on the recovered counters if no potential attacks happen.

By introducing an extra register of N_{wb}, we can now detect the potential replay attacks, but still cannot locate the exact tampered data block. However, the probability of this attack occurring is extremely low (the *dirty address queue* can contain at most 42 dirty counters, whose size occupies only 0.01% of a 16GB NVM). What's more, such replay attack is unable to be located only when the system crashes, which further reduces the probability. We believe that adding more persistent registers to record all the dirty counter addresses in *dirty address queue*, and the update times of each dirty counter cache can help us to locate the tempered data blocks, with the cost of higher hardware requirements.

5 EVALUATION AND RESULTS

We model the hardware design in the cycle-accurate simulator Gem5 [2]. The simulated system consists of x86-64 out-of-order processors running at 3GHz. Each core has a private L1 I/D cache of 32KB (latency=2 cycles), 2-way set associative and a shared L2 cache of 256KB (latency=20 cycles), 8-way set associative. The processor includes shared 128KB (latency=32 cycles), 8-way set associative counter cache and Merkle Tree cache at L2 cache level. All caches have 64B blocks and use LRU replacement. Without loss of generality, we model PCM technologies with read/write latency of 60ns/150ns [8] and 16GB capacity. The read/write queue in the memory controller is 32/64 entries. We assume the overall AES encryption latency to be 72 ns [22] and the HMAC computation based on SHA-1 to be 80-cycle latency [18, 19]. The HMAC is 128-bit codewords, thus the Merkle Tree is 4-ary with 12 levels. The look-up latency of *dirty address queue* is 32 cycles. We use benchmarks from the SPEC2006 suite [16]. For each simulation, we simulate for 500 million instructions after fast-forwarding to representative regions. All experiments are single-thread and single-core. Here are the different designs we use in our evaluation:

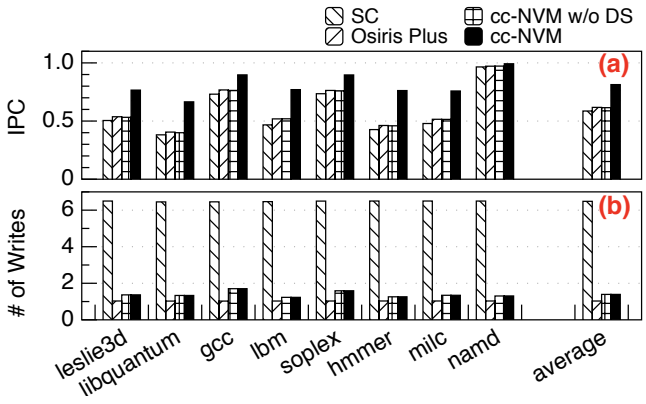

Figure 5: System IPC (a) and NVM write traffic (b) for different designs. Results are normalized to w/o CC.

w/o Crash-Consistency (w/o CC) is secure NVM without crash-consistency. It only writes to memory dirty evictions from cache. And this is our normalized base.

Strict Consistency (SC) enforces the data block and the corresponding metadata (e.g., the counter and tree nodes in MT) to be atomically written to NVM, with root consistently updated in TCB. The atomic mechanism is based on the persistent registers [9].

Osiris Plus is an optimized version of the Osiris that eliminates the need for evicting dirty counter blocks at the cost of extra online checking to recover the most recent counter value [9].

cc-NVM w/o DS is our base solution without deferred spreading (DS). It eliminates the need for atomic updates for every data block while minimizing the performance and write traffic overheads.

cc-NVM is an optimized version that only updates till the cached Merkle Tree node instead of till the root.

The update times is limited to 16 for Osiris Plus, cc-NVM w/o DS and cc-NVM. Besides, we use 64-entry *dirty address queue* for cc-NVM w/o DS and cc-NVM.

5.1 System performance

Figure 5 (a) illustrates the impact of different designs on system performance in instructions per cycle (IPC) over different benchmarks. We have the following observations: (1) SC, Osiris Plus and cc-NVM w/o DS shows very close but lower system performance compared to cc-NVM. To guarantee the correct recovery, all of them need to ensure the consistent updates between the write-back data and the root node in Merkle Tree. Therefore, only when the root node is updated can the data blocks be forwarded to the write pending queue. Instead, for cc-NVM scheme, write-back data can be forwarded to the write pending queue so long as an accessed metadata is cached

in TCB. (2) We also observe that Osiris-plus performs slightly better than cc-NVM w/o DS, this is because Osiris Plus eliminates the write traffic for Merkle Tree node updates. (3) The performance of cc-NVM is still lower than the baseline. Specifically, it reduces IPC by 18.7% on average. Since in cc-NVM, all write-back data needs to wait until all corresponding secure metadata addresses are put into the *dirty address queue*.

5.2 NVM write traffic

Figure 5 (b) shows the comparison of memory write traffic incurred by different designs over baseline w/o CC. Strict consistency has the most number of writes. Since a BMT is a multi-level tree of hashes with counters as its leaf nodes, in strict consistency, all the BMT nodes that lie on the branch of the modified counter on every write-back need to be written back. In our simulation, a 16 GB NVM with a 12-level 4-ary BMT requires 12 atomic BMT updates on every write-back (the BMT root is updated on the TCB, whereas 10 internal path nodes and the leaf-level counter are updated in the NVM). This results in high memory write traffic, which negatively impacts NVM lifetime. Both cc-NVM and cc-NVM w/o DS shows similar memory write traffic. Specifically, their write traffic is 39% higher than the baseline. The extra write traffic is introduced when the updated tree nodes in Merkle Tree is flushed to NVM. Osiris Plus has less write traffic than cc-NVM. This is because it doesn't have to persist the tree nodes of the Merkle Tree. However, we notice that the performance of cc-NVM is higher than Osiris Plus, despite the fact that cc-NVM incurs higher write traffic. The reason being, (1) all extra metadata write traffic is incurred by data write-back, which is out of the critical path of the CPU execution. (2) the NVM bandwidth is not the bottleneck in our tests.

5.3 Sensitivity: Trigger Conditions

There are two parameters we can change to affect the epoch length. One is the number of entries (M) in *dirty address queue*, the other is the limit of update times (N) for the dirty cache blocks in metadata cache. As shown in the Figure 6, the larger number of M and N may result in longer epoch, thus improving the system performance and reducing the number of writes to NVM. As we can see in Figure 6 (a), it has little effect on performance and write traffic when the N is larger than 32 (and the M is set 64 in these experiments). Because the other two trigger conditions are dominant for state transition. In Figure 6 (b), the N is set 16. The number of entries in write pending queue limit the number of entries in *dirty address queue*. Thus it must be less than 64. cc-NVM achieves less write traffic and better performance with larger M. While when the M is larger than 48, the effect of M slows down. This is because the other two trigger conditions play major roles.

6 CONCLUSIONS

cc-NVM is a low overhead crash-consistency mechanism for encrypted and authenticated NVM. cc-NVM employs epoch-based consistent BMT with deferred spreading to achieve high-performance and write-efficiency. After crashes, cc-NVM is capable to detect/locate attacks in NVM and continue normal secure protection. In comparison to state-of-the-art secure NVM Osiris Plus [9] that guarantees crash-consistency, cc-NVM improves IPC by 20.4% on average,

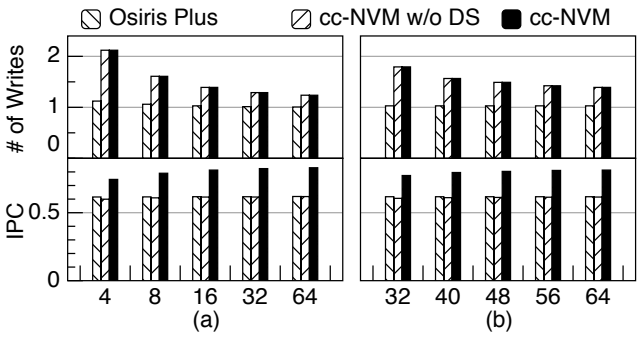

Figure 6: (a). Impacts with varying update times limit (N). (b). Impacts with varying number of entries (M) in *dirty address queue*. Results are normalized to w/o CC.

while only incurring extra NVM write traffic by 29.6% with the ability to locate attacks after crashes.

REFERENCES

[1] 2015. 3D XPoint. "https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/".
[2] Nathan Binkert et al. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* (2011).
[3] B. Gassend et al. 2003. Caches and hash trees for efficient memory integrity verification. In *HPCA-9*. IEEE.
[4] Gururaj Saileshwar et al. 2018. Morphable Counters : Enabling Compact Integrity Trees For Low-Overhead Secure Memories. In *MICRO*.
[5] H. Mao et al. 2017. Protect non-volatile memory from wear-out attack based on timing difference of row buffer hit/miss. In *DATE*.
[6] J. Lee et al. 2016. Reducing the Memory Bandwidth Overheads of Hardware Security Support for Multi-Core Processors. *IEEE Trans. Comput.* (2016).
[7] Kültürsay et al. 2013. Evaluating STT-RAM as an energy-efficient main memory alternative. In *ISPASS*. IEEE.
[8] Lee et al. 2009. Architecting phase change memory as a scalable dram alternative. In *ACM SIGARCH Computer Architecture News*. ACM.
[9] Mao Ye et al. 2018. Osiris: A Low-Cost Mechanism to Enable Restoration of Secure Non-Volatile Memories. In *MICRO*.
[10] Pengfei Zuo et al. 2018. SecPM: a secure and persistent memory system for non-volatile memory. In *HotStorage*. USENIX.
[11] S. Swami et al. 2016. SECRET: Smartly encrypted energy efficient non-volatile memories. In *Design Automation Conference (DAC)*. IEEE.
[12] S. Swami et al. 2018. ARSENAL: Architecture for Secure Non-Volatile Memories. *IEEE Computer Architecture Letters* (2018).
[13] T. S. Lehman et al. 2016. PoisonIvy: Safe speculation for secure memory. In *MICRO*. IEEE Press.
[14] Y. Lu et al. 2014. Loose-Ordering Consistency for persistent memory. In *ICCD*.
[15] Y. Lu et al. 2015. Blurred persistence in transactional persistent memory. In *MSST*.
[16] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* (2006).
[17] Sihang Liu et al. 2018. Crash Consistency in Encrypted Non-volatile Main Memory Systems. In *HPCA*. IEEE.
[18] Joydeep Rakshit et al. 2017. ASSURE: Authentication Scheme for SecURE energy efficient non-volatile memories. In *Design Automation Conference (DAC)*. IEEE.
[19] Brian Rogers et al. 2007. Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly. In *MICRO*. IEEE Computer Society.
[20] A. M. Rudoff. 2016. Deprecating the pcommit instruction. https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction. 2016.
[21] G. Edward Suh et al. 2003. Efficient memory integrity verification and encryption for secure processors. In *MICRO*. IEEE Computer Society.
[22] Shivam Swami et al. 2018. ACME: Advanced counter mode encryption for secure non-volatile memories. In *Design Automation Conference (DAC)*. IEEE.
[23] Shivam Swami et al. 2018. STASH: Security architecture for smart hybrid memories. In *Design Automation Conference (DAC)*. IEEE.
[24] Chenyu Yan et al. 2006. Improving cost, performance, and security of memory encryption and authentication. In *ISCA*. IEEE Computer Society.
[25] Vinson Young et al. 2015. DEUCE: Write-efficient encryption for non-volatile memories. *ACM SIGARCH Computer Architecture News* (2015).