Persist Level Parallelism: Streamlining Integrity Tree Updates for Secure Persistent Memory

Alexander Freij*, Shougang Yuan†, Huiyang Zhou‡
Dept. of Electrical & Computer Engineering
North Carolina State University
*atfreij@ncsu.edu, †syuan3@ncsu.edu, ‡hzhou@ncsu.edu

Yan Solihin
Dept. of Computer Science
University of Central Florida
Yan.Solihin@ucf.edu

***Abstract*—Emerging non-volatile main memory (NVMM) is rapidly being integrated into computer systems. However, NVMM is vulnerable to potential data remanence and replay attacks. Memory encryption and integrity verification have been introduced to protect against such data integrity attacks. However, they are not compatible with a growing use of NVMM for providing crash recoverable persistent memory. Recent works on secure NVMM pointed out the need for data and its metadata, including the counter, the message authentication code (MAC), and the Bonsai Merkle Tree (BMT) to be persisted atomically. However, memory persistency models have been overlooked for secure NVMM, which is essential for crash recoverability.**

In this work, we analyze the invariants that need to be ensured in order to support crash recovery for secure NVMM. We highlight that by not adhering to these invariants, prior research has substantially under-estimated the cost of BMT persistence. We propose several optimization techniques to reduce the overhead of atomically persisting updates to BMTs. The optimizations proposed explore the use of pipelining, out-of-order updates, and update coalescing while conforming to strict or epoch persistency models, respectively. We evaluate our work and show that our proposed optimizations significantly reduce the performance overhead of secure crash-recoverable NVMM from 720% to just 20%.

***Index Terms*—persistency, security, integrity tree update, persist-level parallelism**

I. Introduction

Non-volatile main memory (NVMM) is coming online, offering non-volatility, good scaling potential, high density, low idle power, and byte addressability. A recent NVMM example is Intel Optane DC Persistent Memory, providing a capacity of 3TB per socket [22]. Due to non-volatility, data may remain in main memory for a very long time even without power, exposing data to potential attackers [8]. Consequently, NVMM requires memory encryption and integrity protection to match the security of DRAM (which we refer to as *secure NVMM*), or to provide secure enclave environment. Furthermore, it is expected that NVMM may store persistent data that must provide *crash recoverability*, a property where a system can always recover to a consistent memory state after a crash. Crash recoverability property offers multiple benefits, such as allowing persistent data to be kept in memory data structures instead of in files, and as a fault tolerance technique to reduce checkpointing frequency [1], [14], [23], [24], [48]. Finally, some applications have emerged that need to run on secure enclave and yet require persistency and crash recovery, such as a shadow file system [19].

This work was supported in part by grant 1717550 from NSF, and by UCF. The work was initially started under the direction of Solihin while at NCSU and further co-directed by Zhou and Solihin at NCSU and UCF, respectively.

Crash recovery of data with NVMM is achieved through defining and using memory persistency models. However, there has not been a systematic study examining how secure NVMM can support crash recovery on persistency models. Supporting persistency models on secure NVMM incurs two new requirements: ① *the correct plaintext value of data must be recovered*, and ② *data recovery must not trigger integrity verification failure for a given persistency model*. To meet these requirements, the central question is what items must persist together, and what persist ordering constraints are there to guarantee the above requirements? No prior studies have provided a complete answer. Liu et al. pointed out that counters, data, and message authentication codes (MACs) must persist atomically [33], but ignored the Merkle Tree for integrity verification. Awad et al. pointed out that Merkle Tree must also be persisted leaf-to-root [4], but did not specify ordering needed for persistency models.

The focus of this work is to comprehensively analyze the persist and persist ordering requirements required for correct crash recovery on secure NVMM. Getting this analysis right is important. Not only does it affect correctness (i.e., whether the above crash recovery requirements are met), but it also affects the accurate performance overheads estimation and the derivation of possible performance optimizations. For example, one property missed by prior work is that leaf-to-root updates of Bonsai Merkle trees (BMT) must follow persist order, otherwise crash recovery may trigger integrity verification failure at system recovery. Obeying this ordering constraint, we found that the overheads of crash recoverable strict persistency (SP) is about $30\times$ slowdown, which is more than one order of magnitude higher than previously reported slowdown.

In this paper, we analyze and derive invariants that are needed to ensure correct crash recovery (i.e., correct plaintext value is recovered and no integrity verification failure is triggered). Then, to reduce the performance overheads, we propose performance optimizations, which we refer to as *persist-level parallelism*, or PLP, that comply with the invariants for strict and epoch persistency (EP) models. For SP, we found that pipelining BMT updates is an effective PLP optimization, which brings down the performance overheads from 7.2× to 2.1× when protecting non-stack regions, compared to a secure processor model with write back caches but not supporting any persistency model. We then analyze EP where persist ordering within an epoch is relaxed, but enforced across epochs. Under EP, two more PLP optimizations were enabled besides pipelining: out-of-order BMT update and BMT update coalescing. These two optimizations reduce overheads to 20.2%.

To summarize, the contributions of this paper are:

- To our knowledge, this is the first work that fully analyzes crash recovery correctness for secure NVMM, and formulates crash recovery invariants required under different persistency models.
- For strict persistency, we propose a new optimization for pipelining BMT updates.
- For epoch persistency, we propose two new optimizations: out-of-order BMT updates and BMT update coalescing.
- We point out that, many techniques in prior studies did not completely guarantee crash recovery and hence substantially underestimated its performance overheads.
- An evaluation showing that our proposed PLP optimizations above significantly reduce the performance overhead of secure NVMM.

The remainder of the paper is organized as follows. Section II presents the background and related work. Section III formulates the invariants to be ensured in order to support crash recovery for secure NVMM. Section IV details four BMT update models, including the baseline used for evaluation and the three proposed ones. Section V discusses our hardware architecture. Section VI presents our experimental methodology. Section VII evaluates our proposed update mechanisms, and Section VIII concludes this work.

II. Background and Related Work

Threat Model We assume an adversary who has physical access to the memory system (NVMM and system bus), e.g. through ownership, theft, acquisition after system disposal, etc. Similar to the incidence of recovering sensitive data from improperly disposed used hard drives [41], [58], data remanence in NVMM extends such vulnerabilities to data in memory [8]. In addition, NVMMs are potentially vulnerable to replay attacks [2] and cold boot attacks [20], [37], which allow malicious entities access to the systems. Similar to prior work [3], [4], [30], [31], [47], we assume that the adversary cannot read the content of on-chip resources such as registers and caches, hence the processor chip forms the trust boundary where trusted computing base (TCB) may be located. All off-chip devices, including main memory and memory bus, are considered vulnerable to both passive (snooping) and active (tampering) attacks. These assumptions are essential to secure processor architecture [9], [15], [51], [54], [57], [60], [61].

Memory Encryption The goal of memory encryption is to conceal the plaintext of data written to the off-chip main memory [29], [32], [44], [53], [67] or sent to other processor chips [42], [44], [64]. Counter mode encryption [52], [60], [61] is commonly used for this purpose. It works by encrypting a counter to generate a pseudo one time pad (OTP) which is XORed with the plaintext (or ciphertext) to get ciphertext (or plaintext). To be secure, pads cannot be reused, and hence the counter must be incremented after each write back (for temporal uniqueness) and concatenated with address to form a seed (for spatial uniqueness). Counters may be monolithic (as in Intel SGX [12], [18]) or split (as in Yan et al. [60]). Split counter co-locates a per-page major counter and many per-block minor counters on a single cache block, and each cache block is represented by the concatenation of a major and a minor counter. Due to its much lower memory overhead (1.56% vs. 12.5% with monolithic counter [60]), counter cache performance increases and the overall decryption overhead decreases. Hence, we assume the use of a split counter organization for the rest of the paper.

Memory Integrity Verification Memory encrypted using counter mode encryption is vulnerable to a *counter replay attack* which allows the attacker to break the encryption [60], hence memory integrity verification is needed not only to protect data integrity, but also to protect encryption from trivial cryptanalysis [39], [65]. Data fetched from off-chip memory must be decrypted and its integrity verified when it arrives on chip. In multiprocessors, data supplied from other processor chips also need to be verified [42], [44]. Early memory integrity protection relied on Merkle Tree covering the entire memory [16] with on chip tree root. When using counter mode encryption, Rogers et al. proposed Bonsai Merkle Tree (BMT) [43] that employs stateful MACs to protect data, leaving a much smaller and shallower tree covering only counters. A stateful MAC uses data, address, and counter as input to the MAC calculation; any modification to any MAC input or the MAC itself becomes detectable. Since it is sufficient to have one input component with freshness protection, BMT only needs to cover counters. Intel SGX adopted this observation to design a similar stateful MAC approach to construct a counter tree that combines counters and MACs [18].

Memory Persistency Memory persistency is defined to allow the reasoning of crash recovery for persistent data [1], [6], [7], [11], [13], [25], [27], [38], [40], [59]. It defines the ordering of stores as seen by a crash recovery observer [35], [38], pertaining when a store *persists* (i.e. becomes durable) with respect to other stores of the same thread. Since visibility to crash recovery observer and other threads may be intertwined, it is sometimes coupled with memory consistency models.

The most conservative model, *strict persistency* (SP) requires that persists follow the sequential program order of stores [38]. While providing simple reasoning, SP does not allow any overlapping or reordering of persists, limiting optimization opportunities in the system and incurring high performance overheads. More relaxed persistency models include *epoch persistency* (EP) and *buffered epoch persistency*

(BEP) [38], as well as *lazy persistency* [1]. With EP/BEP, programmers define regions of code that form *epochs* [17], [26]. Persists within an epoch can be reordered and overlapped, but persists across epochs are strictly ordered using persist barriers, which enforce that persists in an older epoch must complete prior to the execution (or completion) of any persist from a younger epoch. On top of a persistency model, crash recovery often requires the programmer to define atomic durable code regions [10], [13], [36], [45], [49], [63].

WPQ and Metadata Caches Modern processors utilize a *write pending queue* (WPQ) in the memory controller (MC) [45]. System features such as Asynchronous DRAM Refresh (ADR) adds WPQ to the persistence domain by requiring that the contents of the WPQ are flushed to NVMM when a crash occurs [45], making WPQ the point of persistence for stores.

Counters, MACs, and Merkle Tree nodes may be placed in the last level cache [43] or in their own metadata caches [16], [43], [50], [51], [60], [61]. Metadata caches may be unified for all metadata types [46], [55] or separate [30], [62]. Our models assume separate metadata caches.

Secure NVMM for Crash Recovery Data remanence vulnerability for DRAM as data may persist for weeks under very low temperature [20], [37]. The vulnerability is much worse with NVM since data is retained for years, hence self-encrypting memory has been proposed [8]. However, NVM will likely host persistent data supporting crash recovery, requiring integrating memory encryption and integrity verification with memory persistency. This has been explored only recently. Swami et. al [55] proposed co-locating data, counters, and MAC, to make it easier to atomically persist them together. Liu et al. [33] proposed a similar approach, plus an alternative approach of using the MC as a gathering point for atomic persistence. Awad et al. [4] looked at persisting data, counters, and BMT, but did not address persistency models and persist ordering. Zuo et. al [68] proposed coalescing counters for persisting counter cache data, but did not discuss counter integrity verification. Liu et. al [34] optimized backend memory operations (BMO) including encryption, integrity protection, compression, and deduplication and proposed parallelized execution and pre-execution with compiler support to reduce the BMO overhead. Persistency models and persist ordering of BMT updates were not discussed. Finally, in non-NVM context, Saileshwar et. al [47] and Taassori et. al [56] proposed mechanisms to reduce the integrity tree size. However, while shallower, the fundamental bottleneck of having to update BMT from leaf-to-root in persist order remains, which is what is addressed in this paper.

III. Correctness of Crash Recovery

Supporting crash recovery requires three levels of mechanisms. At the highest level is the programmer specifying durable atomic region, which allows a group of stores to persist together or not at all. With Intel PMEM, building such a region needs to rely on creating and keeping undo/redo logging in software. Building such a region requires the next level of mechanism (*persistency model*), which specifies the ordering of the persistence of stores with respect to program order, such as strict persistency, epoch persistency, etc. Each persistency model relies on the next level mechanism which must ensure that each store, if it persists, must be recoverable to its original plaintext value and must not trigger integrity verification failure. It is the last level mechanism that our work seeks to provide.

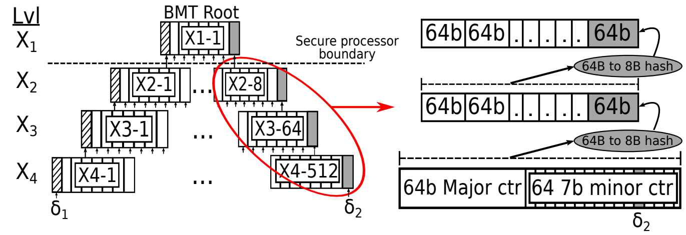


Fig. 1. An example illustrating two BMT updates with their update paths. Persist δ_1's path is shown as striped pattern (X4-1, X3-1, X2-1, X1-1) while δ_2's update path is shown in the grey color (X4-512, X3-64, X2-8, X1-1). Each MAC takes a 64-byte input and outputs a 64b hash value.

In this section, we formulate the invariants to be ensured in order to support crash recovery for secure NVMM. The system we assume is one with volatile on-chip caches and a persistent domain that includes NVMM and the WPQ inside the MC. Our analysis focuses on a system with counter-mode memory encryption along with MAC and BMT integrity verification. Counters, MACs, and BMT nodes are cacheable and can be lost with the loss of power, except the BMT root which is always stored persistently on chip. Recovering from a crash requires recomputing the BMT root and validating it against the stored root. We discuss Intel SGX MEE later in the paper.

Suppose that plaintext P at address A is encrypted using counter γ and private key K to yield ciphertext C, i.e., $C = E_K(P, A, \gamma)$ and necessarily the decryption follows $P = D_K(C, A, \gamma)$. Suppose also that M represents a message authentication code for C, i.e., $M = MAC_K(C, A, \gamma)$. Finally suppose that BMT covers all counters and has a root R. We define BMT update path as follows:

Definition 1: **BMT update path** is the path of nodes from a leaf node (i.e., one encryption page) to the root of BMT.

Fig. 1 shows an example with two persists that generate updates to an 8-ary BMT. Update δ_1 affects all $\frac{1}{8}$ parts of nodes shown in stripes, while update δ_2 affects all $\frac{1}{8}$ parts of nodes shown in grey. The update paths intersect at the BMT root and different parts of it are modified. Note that while all update paths necessarily intersect at the root, they may intersect earlier.

Definition 2: **Common Ancestors** of two persists are nodes in the BMT tree that appear in the BMT update paths of both persists. The **Least Common Ancestor** (LCA) is a common ancestor that is at the lowest-to-leaf level compared to all other common ancestor nodes.

In the example in Fig. 1, the common ancestor consists of only the BMT root, hence the BMT root is also the LCA. However, if another persist causes an update at node $X4$-2,

TABLE I
RECOVERY FAILURE CASES DUE TO PERSIST FAILURE

C	γ	M	R	Outcome
$\surd$	$\surd$	$\surd$	$\times$	BMT (verification) failure
$\surd$	$\surd$	$\times$	$\surd$	MAC (verification) failure
$\surd$	$\times$	$\surd$	$\surd$	Wrong plaintext, BMT&MAC failure
$\times$	$\surd$	$\surd$	$\surd$	Wrong plaintext, MAC failure

TABLE II
RECOVERY FAILURES DUE TO MEMORY TUPLE ORDERING VIOLATION. SUPPOSE THAT $\alpha_1 \to \alpha_2$ IS ENFORCED BY PERSISTING C_1 BEFORE C_2. ALL MEMORY TUPLE COMPONENTS OF α_1 HAVE BEEN PERSISTED EXCEPT FOR THE ONE THAT SUFFERS FROM ORDERING VIOLATION, WHERE α_2'S TUPLE COMPONENT WAS PERSISTED INSTEAD.

Violating $\gamma_1 \to \gamma_2$	Plaintext P_1 not recoverable
Violating $M_1 \to M_2$	MAC (verification) failure for C_1 and C_2
Violating $R_1 \to R_2$	BMT (verification) failure for C_1

then this update and δ_2 share $X3$-1 and $X1$-1 as common ancestors, with $X3$-1 being the LCA.

We also define a memory tuple as a collection of items that are needed to crash recover a datum:

Definition 3: Secure memory transforms an on-chip plaintext data P at block address A to a **memory tuple** of (C, γ, M, R) when data is persisted to main memory, and vice versa when persisted data is read from main memory.

The memory tuple represents the totality of transformation of a block when it is written back (out of the last level cache or LLC) to off-chip memory, and we claim that each tuple item must be available in order to recover data correctly, and failure to persist any item(s) in the tuple results in a crash recovery problem:

Invariant 1: **Crash Recovery Tuple Invariant**. In a secure memory with counter-mode encryption and MAC/BMT verification, in order to recover a datum P that was persisted in memory, its entire memory tuple (C, γ, M, R) must have been persisted as well.

To illustrate this, suppose that a plaintext value P_o is changed to a new value P_n. The memory tuple for the block then must change from $(C_o, \gamma_o, M_o, R_o)$ to $(C_n, \gamma_n, M_n, R_n)$. If some tuple item was not persisted, for example M_n, post-crash, $(C_n, \gamma_n, M_o, R_n)$ is recovered. In this case, the correct plaintext is recovered but MAC verification fails because the old MAC (M_o) fetched from memory mismatches with $MAC_K(C_n, A, \gamma_n)$. If instead γ_n was not persisted, since $P_n \neq D_K(C_n, A, \gamma_o)$, the correct plaintext is not recovered. Not only that, since γ_o is input to MAC and BMT verification, both verification mechanisms fail as well. Table I lists the outcomes of not persisting one or more of the memory tuple.

Note that the crash tuple invariant (Invariant 1) specifies the necessary and sufficient condition for recovering data post crash. It does not specify exactly "when" tuple items must be persisted with respect to the data persist; this depends on the crash recovery expectation of the program and the persistency model being assumed.

So far we have discussed the crash recovery correctness for a single data persist. To support crash recovery, programmers must reason about not just a single persist, but multiple persists and the relative ordering between them. In this case, we assume that if there is possibility that the crash recovery observer reads the persistent memory state between two persists, then the two persists must be ordered. Now suppose that there are two ordered persistent stores (persists) α_1 and α_2 to the different blocks. For the memory tuples of these different blocks, it is possible that these blocks may modify the same counter block, the same MAC block, and definitely the same BMT root. If the persist order of memory tuples is not followed, recoverability is problematic. For example, suppose that $\alpha_1 \to \alpha_2$ but $R_2 \to R_1$, which means that the BMT root is updated by the second persist before by the first persist. If a crash occurs prior to either of them or after both of them, recoverability is not jeopardized. But at other points, recovery can fail. For example, suppose that a crash occurs after α_1 and R_2 persist but before α_2 and R_1 persist. Post crash, BMT verification failure occurs due to the root not reflecting the persist of α_1. In other words:

Invariant 2: **Persist Order Invariant**. Suppose that α_1 happens before α_2 in program order. If the crash recovery observer may read out the persistent state between α_1 and α_2, then α_2 must follow α_1 in persist order, i.e. $\alpha_1 \to \alpha_2$. If $\alpha_1 \to \alpha_2$ in persist order, then for correct crash recovery, the following must hold: $(C_1, \gamma_1, M_1, R_1) \to (C_2, \gamma_2, M_2, R_2)$ in persist order, i.e. the persist order of each respective memory tuple items must follow the order of data persists.

Note that the persist order depends on the persistency models. For SP, every persist is ordered with respect to others and Invariant 2 applies to each pair of persists. For EP, Invariant 2 applies only to stores from different epochs. Persists from the same epoch are unordered, which gives a rise to optimization opportunities discussed in Section IV.

Implications There are several consequences of Invariant 2. Ordering violation triggers recovery failures as listed in Table II. Current persistency model specifications are incomplete for secure NVMM as they only enforce ordering of data persists (e.g. $C_1 \to C_2$). Persist barrier (such as `sfence`) needs to expand its semantics to also include other tuple components.

In addition, mechanisms or optimizations that may reorder tuple updates violate Invariant 2. For example, suppose that C_2 is available early (due to prediction): pre-computing γ_2 or M_2 carries the risk of them being evicted from the metadata caches earlier than γ_1 or M_1, hence violating the invariant. Furthermore, two persists α_1 and α_2 could incur different latencies to update their respective BMT paths because some BMT nodes may be found on chip while others need to be fetched from main memory. Without an explicit mechanism to enforce the ordering of BMT path updates, Invariant 2 is likely violated often. To our knowledge, our work is the first to identify the need to order BMT (and tuple) updates. Finally, naive mechanisms to enforce persist ordering impose a very high cost that scales with the size of BMT, exposing *BMT* *updates as the primary performance bottleneck* for a secure NVMM. Upon eviction of a block from LLC, the data, its counter, and MAC are updated and sent to the MC, but they must wait until BMT root is updated before the persist can be considered successful. For example, assuming a hash latency of 80 processor cycles [30], updating a 9-level BMT incurs 720 processor cycles for one persist.

IV. Streamlining BMT Updates

In this section, we explore how BMT update performance due to persists can be improved. Performance optimization techniques that are possible depend on ① no violation against invariants discussed in the previous section, and ② the persistency model that is assumed. We collectively refer to the key methods as persist-level parallelism (PLP): pipelining, out-of-order updates, and coalescing.

A. Strict Persistency

1) Baseline Atomic Persist Mechanism: Following Invariant 1, for each memory update, we need to ensure that all memory tuple components also persist. Due to the write-back cache, the eviction order of dirty blocks may be different from the program order. Therefore, with SP, one way to satisfy the invariant is to atomically persist the tuple generated by each store, which results in write-through cache behavior. To achieve this, we devise a *2-step persist* (2SP) mechanism. Similar to [33], 2SP relies on the WPQ of the MC as persist gathering point. 2SP consists of two steps: the first step involves gathering and locking persist memory tuple components in the WPQ (while flagged as incomplete), while the second step flags the completion of the persist and releases tuple components to memory. A persist is marked completed when the WPQ receives its updated ciphertext, updated counter, MAC, and acknowledgement that the BMT root has been updated. Once completed, the blocks are allowed to drain from the WPQ to the NVMM. On power failure, any incomplete flagged blocks are considered not persisted and invalidated. Since the persistence of the counter and MAC is straightforward and not expensive, we will focus the rest of the discussion on the expensive BMT update.

To illustrate the mechanism, suppose that two persists are initiated, as shown in Fig. 1. Fig. 2 shows the sequence of persists of memory tuples due to the two persists, in the baseline persist mechanism. For persist δ_1, ciphertext C_1, counter γ_1, MAC M_1 are persisted. A new value of counter γ_1 is needed for the BMT update path starting from leaf of BMT $X4$-1, which in turn is needed to update BMT node $X3$-1, and so on, until BMT root $X1$-1 is updated. When ciphertext C_1, counter γ_1, and MAC M_1 are completed and BMT root is updated, δ_1 is considered completed, after which persist δ_2 can commence. It is clear that even though intermediate nodes in the BMT update path do not need to persist (only the leaves and root need to persist), the critical path is due to their sequential updates.

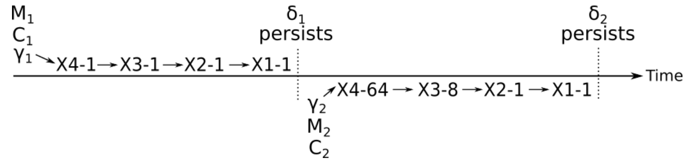


Fig. 2. The timeline of two data persists and their memory tuple persists.

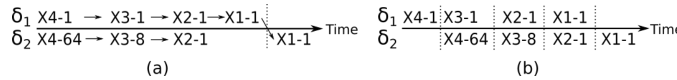


Fig. 3. The timeline of (a) out-of-order BMT updates with in-order BMT root updates, and (b) pipelined updates with in-order common ancestor (including BMT root) updates.

2) PLP Mechanism 1: Pipelining BMT Updates: While the baseline persist mechanism described in Section IV-A1 is correct, it suffers from high overheads. Each node in the BMT update path must wait until the previous node has been calculated. In order to improve this situation, recall that the Persist Order Invariant (Invariant 2) only requires that the BMT root update follows the persist order. This means that it is possible to update BMT nodes out of order, as long as the root is still updated in persist order. This is illustrated in Fig. 3(a), where update paths of persist δ_1 and persist δ_2 are updated out of order but updates to BMT root are kept in persist order.

While out of order non-root updates are best for performance, it is difficult to avoid write-after-write (WAW) hazards if two persists' BMT update paths intersect at more than just the BMT root. To avoid WAW without much complexity, we design a more restrictive version of the optimization, namely *pipelined BMT update*. With a pipelined update, a younger persist is allowed to update a certain level of BMT only when an older persist has completed its update of the same level BMT node. This is illustrated in Fig. 3(b). The pipelined update optimization ensures that if two persists have common ancestor nodes, they will still be updated in persist order.

Note that as the memory grows bigger, the BMT will have more levels and hence more pipeline stages. Thus, one attractive feature of pipelined BMT updates is that with larger memories, the degree of PLP increases and pipelined BMT updates becomes even more effective versus non-pipelined updates.

B. Epoch Persistency

With EP, two persists in the same epochs do not have persist ordering constraints; persists only need to be ordered across separate epochs. This fact allows the write-back cache to reduce the write traffic and also gives us opportunities to optimize BMT updates. We make a stronger assumption on EP compared to that in literature: Nalli et. al [36] assert that 75% of epochs update one 64B cache line, where we assume a minimum of one store per epoch. Specifically, we assume that crash recovery does not depend on the transient persistent state within an epoch while an epoch is executing. Instead,

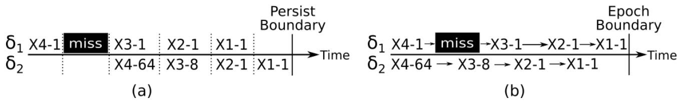


Fig. 4. The timeline of two data persists with (a) in-order pipelining and (b) out of order updates.

crash recovery depends only on the persistent state at an epoch boundary. This assumption requires that any actions performed by an epoch that were not completely persisted prior to crash must be re-executable. This assumption is reasonable, because epochs are usually components of a durable transaction, and durable transactions can be re-executed if they fail.

1) PLP Mechanism 2: Out-of-Order BMT Updates: Invariant 2 applies to two persists that are ordered, i.e. in EP, they belong to two different epochs. It does not specify how to treat two persists that are not ordered, such as those belonging to the same epoch. The question then arises whether two unordered persists can be performed out of order (OOO), and if so, to what extent and whether there are any constraints that need to be observed.

Before discussing them further, let us first discuss the potential benefit of OOO. OOO BMT updates have a much better performance potential than (in-order) pipelining for two reasons. First, it can hide the BMT cache miss latency as illustrated in Fig. 4. Fig. 4(a) shows a case where persist δ_1 is attempting to update the BMT, but suffers a cache miss on BMT node $X4$-1. This introduces bubbles in the in-order BMT update pipeline, and persist δ_2 is consequently delayed, therefore it cannot update $X4$-64 until $X4$-1 is updated. Fig. 4(b) illustrates that with OOO, both updates can occur in parallel, with δ_2 not being delayed by the cache miss that δ_1 must wait for. Therefore, OOO can achieve a higher degree of PLP compared to in-order pipelining. Second, OOO BMT updates enable us to use pipelined MAC units to improve the throughput. The in-order BMT update pipeline has the same number of stages as the levels in the BMT and there is at most one update at each level. Therefore, the throughput of pipelined BMT is limited to one BMT update per n cycles, where n is the MAC latency. In contrast, with OOO, a BMT update can start at every cycle, thereby increasing the throughput to one BMT update per cycle.

Regarding correctness of OOO execution of persists from the same epoch, a concern arises that there may be a write after write (WAW) hazard in the case where two persists have their BMT update paths intersecting at not just the BMT root. The hierarchical nature of BMT dictates that if two BMT update paths intersect, the intersection representing common ancestors manifests as common suffix in the paths, starting from the lowest common ancestor (LCA) node, and then continuing to the LCA's parent, grandparent, etc. until the BMT root. Does updating common ancestor nodes out of order trigger a WAW hazard? We assert that they do not.

In order to prove it, we note that different blocks will cause different counters to be updated. Let us denote the old counter values as γ_{1o} and γ_{2o} and the new values as γ_{1n} and γ_{2n}. The counters correspond to either one BMT leaf node (if the counters are co-located in a block) or two BMT leaf nodes (if the counters are not co-located in a block). In the former, the leaf node is the LCA, while in the latter the LCA is further up the tree. Suppose that persist δ_1 updates the LCA before δ_2. Then, at the end of the LCA update for both persists, the LCA value is $MAC_K(\gamma_{1n}, \gamma_{2n}, \ldots)$. If instead δ_2 updates the LCA before δ_1, the LCA value is also $MAC_K(\gamma_{1n}, \gamma_{2n}, \ldots)$, which is unchanged. Therefore, the final LCA value is the same, and hence the BMT root is also the same. The intermediate LCA value is different when δ_1 or δ_2 update the LCA first. However, in EP, the crash recovery observer does not expect a particular persist order for two persists in the same epoch. Furthermore, Invariant 2 assumes that the crash recovery observer will not read the transient persistent state between the two persists. For the latter case, δ_1 and δ_2 will update different parts of the LCA, hence the same proof holds.

The epoch boundary, however, places constraints on the degree of PLP, as it acts as point of ordering; all persists in the previous epoch must complete prior to any persist in a new epoch can complete. Thus, the higher the number of persists in an epoch, the higher is its potential PLP.

To handle OOO, the 2SP only needs minor modifications. When blocks belonging to persists from the same epoch are written back from the LLC, they are no longer locked in the WPQ. They are allowed to drain to persistent memory as they come. However, the WPQ retains enough state to monitor if the memory tuples of persists of the same epoch have all arrived at the WPQ or not. When they have all arrived, they are marked completed and the epoch is considered complete. On the other hand, blocks from the next future epoch are locked in the WPQ and marked incomplete, until the previous epoch has completed.

2) PLP Mechanism 3: BMT Update Coalescing: Further analysis of BMT updates within an EP model exposes a notable scenario that enables our final optimization. BMT updates within an epoch are likely to involve substantial number of common ancestor nodes, due to spatial locality. While OOO allows updates to BMT to be overlapped and performed out of order, there are still many updates to BMT nodes that occur. These updates can be considered superfluous considering that the same node may be updated multiple times by persists from the same epoch. In our final optimization, we seek to remove superfluous BMT updates by coalescing them.

Fig. 5 illustrates the update order of OOO persists with coalescing. Without coalescing, each persist incurs updating of four BMT nodes, causing a total of 12 updates. With coalescing, persists δ_1 and δ_2 updates are coalesced at their LCA (node $X31$), while δ_3 is coalesced at the LCA at node $X21$. As a result, there are only seven updates to the BMT, which in this example corresponds to 42% reduction in BMT updates. Fewer updates to the BMT reduce the occupancy of the memory integrity verification engine, and hence reduces the latency and improves the throughput of the engine. Furthermore, an equally important benefit to coalescing is the number of writes. Without coalescing, the BMT root is updated three times: with

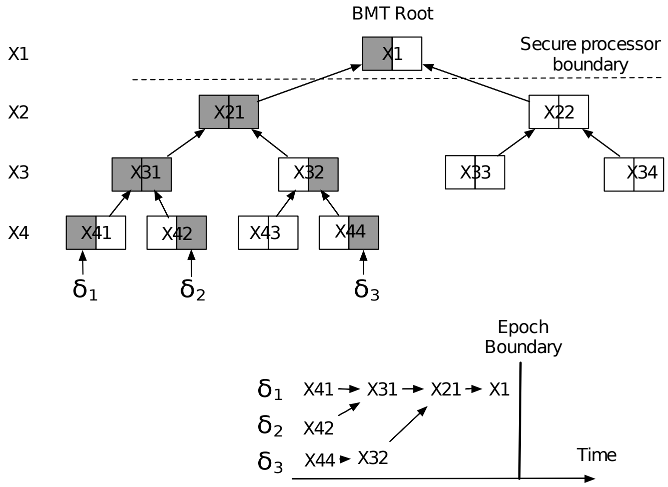


Fig. 5. Example of coalescing BMT updates starting from the lowest common ancestors (LCAs) to the BMT root.

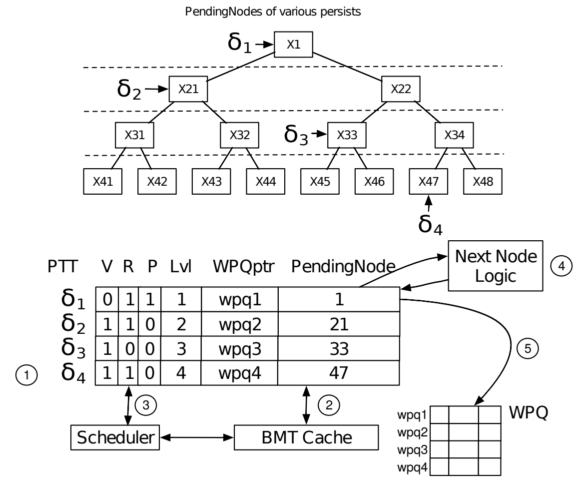


Fig. 6. Example of in-order pipelined update mechanism with Persist Tracking Table (PTT) for SP.

coalescing, it is updated only once.

Coalescing's effectiveness increases with spatial locality. Spatial locality results in nearby blocks being updated. In the best (and also frequent) case, blocks belonging to the same encryption page (a 4KB region) are updated within the epoch. They result in a single counter block being updated multiple times. Without coalescing, each such update generates BMT updates from leaf to root, while with coalescing, there is only one root update, thereby resulting in a substantial saving.

V. Architecture Design

In this section, we propose architecture designs to enable the PLP optimizations. As a baseline architecture, we assume a discrete counter cache [60], BMT cache (*mtcache*) [4], [62], MAC cache [66], and persist-gathering WPQ [33]. These structures suffice if an unoptimized SP model is adhered to. To support our optimizations, additional structures are introduced, specifically schedulers, to retain the persist ordering. These schedulers will contain information that enforces BMT update order by allowing or preventing writes to occur. Each optimization has its own set of conditions for allowing or preventing writes, and will be analyzed next.

A. Strict Persistency: Pipelined BMT Updates

To support our first PLP technique, in-order pipelined BMT updates for SP, we introduce a new structure called *persist tracking table* (PTT) that enforces persist ordering in a SP model.

The PTT interacts with a scheduler that also interacts with the BMT cache and the MC / WPQ. Each entry in the PTT has multiple fields (Fig. 6). The field *Lvl* indicates the level of the BMT that the persist is currently updating, and is used to enforce in-order pipelining by staggering persists on different BMT levels. Fig. 6 shows an example of the PTT with four persist entries. δ_1 is updating level 1 (node $X1$), while δ_2 is updating level 2 (node $X21$), etc. The valid bit V is set when the entry is created and cleared when the persist has updated the BMT root. The ready bit R is set when updating the current BMT node has been completed, and cleared when the update moves on to the next node in the BMT update path. The PTT is managed as a circular buffer using a head and a tail pointer. The persist flag P is set when the BMT root has been updated and the entry can be removed: if the head pointer points to this entry (indicating this entry being the oldest) and the *P* bit is set, then BMT update is considered completed, and both the PTT entry and WPQ entry can be deallocated. The *WPQptr* field points to the corresponding persist entry in the WPQ. The *PendingNode* field indicates the ID/label of the node currently being updated.

In the figure, δ_1 has finished updating the BMT root hence $V = 0$ and $P = 1$. δ_2 and $\delta 4$ have updated their current nodes shown in the *PendingNode* fields, i.e., $X21$ for δ_2 and $X47$ for $\delta 4$, hence $R = 1$. δ_3's R bit is not set yet, either because the BMT node is not yet available for update (e.g. not found in the BMT cache/being fetched from memory), or the update has not completed (e.g., MAC is still being calculated).

The role of the scheduler is to decide when a persist can proceed to updating the next BMT level. To illustrate the working of the scheduler, suppose a new persist request is encountered. An entry is created in the WPQ to hold the data, counter, and MAC to persist. Concurrently, a new PTT entry is also created (Step ①), initialized to point to the corresponding WPQ entry, with the PendingNode labeled with the appropriate leaf BMT node (i.e. MAC of counter block). The valid bit is set, while the ready and persist bits are reset. In Step ②, the BMT cache is looked up for the PendingNode. If found (BMT cache hit), a new MAC is calculated and the node updated. If not found (BMT cache miss), the node is fetched from memory, and the update commences after the node arrives from memory and is verified for integrity. Once the BMT node at the current level is updated, the R bit is set. For the scheduler to allow persist entries to move on to the next BMT levels, it waits until the R bits of these entries are set (Step ③), indicating completion of updates to the current BMT levels. Once the bits are set, the scheduler wakes up the entries to move on to the next BMT levels. The *PendingNode* is input into the Next Node Logic to yield the ID for the next node to update (Step ④).

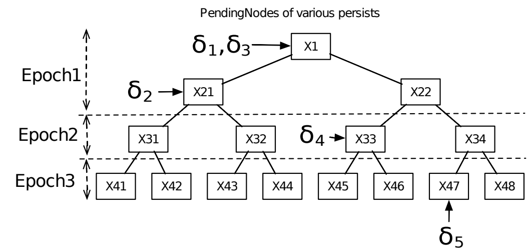
When the oldest entry (δ_1) finishes updating the BMT root, the entry's P bit is set and the WPQ is notified of BMT root update completion (Step ⑤). Afterward, the entry occupied by δ_1 can be released, the head pointer updated, and execution continues. At the WPQ, if BMT root update completion notification is received, and other tuple items are completed (data, counter, and MAC), tuple items are marked as persisted and become releasable to memory.

B. Epoch Persistency: OOO BMT Updates

The previous PTT architecture is not capable of managing BMT updates with EP model with OOO updates of BMT nodes, as it enforces in-order pipelined updates. What is unique with EP is that there are two persist ordering policies: enforced ordering across epochs but not within an epoch. Thus, we split the PTT design into two tables: an *epoch tracking table* (ETT) to track epochs while relegating the PTT to only track persists. Furthermore, coalescing makes the PTT more sophisticated, as it must be able to calculate and track coalescing points of multiple persists. For these reasons, Fig. 7 shows the ETT/PTT split design and also the format of the PTT entries that enable OOO updates and coalescing.

An ETT is a circular buffer maintaining the order of active epochs. An ETT entry has the following fields: *EID* (epoch ID), a valid bit V, a ready bit R (which is set when updates of all persists in the epoch are completed), *Lvl* indicating the lowest BMT level being updated by the epoch, index to the start entry at the PTT (*Start*) and to the end entry at the PTT (*End*). *End* is incremented (wrapped around on overflow) when a new persist from an epoch is encountered. Two special purpose registers are also added: GEC (*global epoch counter*) keeps track of the next epoch ID to allocate to a new epoch, while PEC (*pending epoch counter*) keeps track of the oldest active epoch being processed. In the PTT, each entry is added epoch ID (*EID*) field to identify the epoch a persist belongs to.

Fig. 7 illustrates the tables with an example. There are a total of five persists, with the first three persists from Epoch1, while the fourth and fifth persists are from Epoch2 and Epoch3, respectively. For example, the entry for Epoch1 at the ETT has $Start = 0$ and $End = 2$ to indicate that PTT indices 0..2 contain information of the persists of Epoch1. δ_1, δ_2, and δ_3 are within the same epoch, and hence they perform OOO updates on the BMT root. In the example, δ_3 has updated BMT root $X1$ (hence in the PTT, $P = 1$ and $V = 0$), while δ_1 is working on updating BMT root $X1$ (hence in the PTT, $P = 0$ and $V = 1$). Since δ_3 has persisted, its respective entry can be released from the WPQ assuming all components of the security tuple have been received. δ_2, on the other hand, has not reached BMT level 1 but has finished updating BMT node $X21$ (hence in the PTT, $R = 1$. Since Epoch1 is still



ETT

	EID	V	R	Lvl	Start	End
Epoch1	0	1	0	2	0	2
Epoch2	1	1	0	3	3	3
Epoch3	2	1	1	4	4	4

PTT

	EID	V	R	P	Lvl	WPQptr	PendingNode
δ_1	0	1	0	0	1	wpq1	1
δ_2	0	1	1	0	2	wpq2	21
δ_3	0	0	1	1	1	wpq3	1
δ_4	1	1	0	0	3	wpq4	33
δ_5	2	1	1	0	4	wpq5	47

Fig. 7. Proposed architecture to enable OOO BMT updates and update coalescing within an epoch as well as in-order pipelined BMT updates across different epochs.

working on BMT level 2 node and it is the lowest level that any persist of Epoch1 is working on, in the ETT, Epoch1's $Lvl = 2$. Epoch2 and Epoch3, consisting of one persist each, are updating different nodes ($X33$ and $X47$, respectively) at different BMT levels (level 3 and 4, respectively).

The figure illustrates that we exploit two types of parallelisms: epoch-level as well as persist-level parallelism. Within an epoch, we allow updates to occur OOO. Across epochs, we pipeline updates to the BMT in the epoch order using ETT to track and enforce correctness. The ETT mechanism for pipelining works similarly to the PTT mechanism for pipelining for SP, but with several modifications. First, the ready bit *R* of an epoch is set only when all its persists' ready bits are also set. The *Lvl* of an epoch is determined as the maximum of *Lvl* field of all the persists of the epoch. With this, ETT can ensure that each BMT level can only be updated by persists of a single epoch, which avoids cross-epoch WAW hazards. When all persists of an epoch's are completed within the level(s) that are recorded, an epoch's R bit is set. When all epochs' R bits are set, the epoch-level scheduler is invoked to advance the epochs to the next levels. If an epoch is at level 1 and its completed, the entry can then be deallocated.

Scheduling at the PTT is also modified. In SP, persists update the BMT in a pipelined lockstep fashion. With EP, the persist's EID is used to check which level the persist is authorized to update. In the example in the figure, δ_5 cannot advance to level 3 because Epoch3 is only authorized to update level 4 of the BMT. Apart from epoch-level restriction, each persist can advance to the next level independently of other persists. Hence, assuming the level is authorized, persist-level scheduler allows a persist to advance to the next level whenever $R = 1$ for the persist.

C. Epoch Persistency: Coalescing BMT

To coalesce updates within an epoch, we first need to find the common ancestors. We adopt a BMT node labeling scheme based on the previous work [16]. A unique label is assigned to each BMT node starting from 0 for the BMT root. To find the parent of each BMT node, we subtract one from the label of current node and divided by the arity of the BMT to get the label of its parent. Then we can round this process down until the label 0 to get a list of all its ancestors. The least common ancestor (LCA) between two leaf nodes can be found from the longest prefix match between the two ancestor lists.

Next, we need to decide where to coalesce and how to determine which persists are coalesced together. Consider that it is likely that two persists from the same epoch will share many BMT nodes that are common. Coalescing can occur at any such node. However, the closer to leaf the common ancestor node is, the more effective coalescing become as more updates are eliminated. Therefore, an important principle for update coalescing is to coalesce at LCA whenever possible. The optimal coalescing occurs when the minimum number of updates is achieved. It requires each persist to be compared to every other persist in an epoch, and each pair that has the lowest LCA combined. Then, each combined pair is compared against every other BMT node or pair, and recombined, etc. However, this iterative approach is too costly for hardware implementation. Instead, we opt for paired coalescing, in which we always coalesced the new persist with previous one if it has not been coalesced with other persists.

D. Counter Tree Updates in Intel SGX

Intel SGX utilizes a "counter tree" to verify memory integrity. Similar to BMT, the counter tree does not cover data because it assumes a stateful MAC that protects against spoofing and splicing. The counter tree protects both the integrity and freshness of counters. However, unlike BMT, a counter tree requires the parent counter value to compute the MAC of child counters. As a result, to enable crash recovery, the parent counter value needs to be available and correct in order to compute the correct MAC value. On a store that persists, the tree's entire path from leaf to root nodes must also be persisted, instead of just the tree root.

Therefore, two changes are needed for crash recovery correctness. First, Invariant 1 redefines a memory tuple as consisting of data ciphertext, counter, MAC, and *all* nodes of the counter tree from leaf to root along the update path. Consequently, Invariant 2 expands to include all nodes in the counter tree update path from leaf to root, in contrast to BMT which only requires the tree root to provide crash recovery. This leads to higher costs than BMT. For example, the number of updates that must persist for one store would scale by the height of the counter tree. To enable parallel updates while enforcing these two invariants, we may need to create a shadow copy of the counter tree to ensure atomicity of a single integrity tree update. Such restrictions due to the high inter-level dependence within an SGX Integrity Tree are yet to be explored. In this work, we focus only on BMT due to the extra cost incurred by the counter tree.

VI. Evaluation Methodology

Simulation Model To evaluate our scheme, we built a cycle-accurate simulation model based on Gem5 [5]. Major parameters that we assume are listed in Table III.

TABLE III
SIMULATION CONFIGURATION

Processor Configuration	
CPU	1 core, OOO, x86_64, 4.00GHz
L1 Cache	8-way, 64KB, 64B block
L2 Cache	512KB, 16-way, 64B block
L3 Cache	{1, 2, 4}MB (default 4MB), 32-way, 64B block
WPQ	{4, 8, 16, 32, 64} (default 32 entries)
Metadata Caches	
Counter Cache	{32,64,128,256}KB (default 128KB), 8-way, 64B blk
MAC Cache	{32,64,128,256}KB (default 128KB), 8-way, 64B blk
BMT Cache	{32,64,128,256}KB (default 128KB), 8-way, 64B blk
BMT	9 levels
MAC Latency	{0, 20, 40, 80} processor cycles (default 40) [30], [50]
NVM Parameters	
Memory	8 GB DDR_based PCM, 1200MHz write/read queue: 128/64 entries tRCD/tXAW/tBUSRT/tWR/tRFC/tCL: 55/50/5/150/5/12.5ns [33]
Persistency Model Parameters	
Epoch size	{4, 8, 16, 32, 64, 128, 256} (default 32)
PTT/ETT Size	64 entries (616 bytes) / 2 entries (48 bits)

For all schemes, to verify the integrity of a newly fetched data block, we let decryption and use of data be overlapped with integrity verification [30], [60], [62], [66]. If integrity verification fails, an exception is raised. Separate metadata caches for BMT, MAC, and counters are assumed (parameters in Table III). For strict persistency, we implemented write through caches to persist each store in order to the MC. For the pipelined BMT scheme, we rely on a PTT with 64 entries. To support OOO BMT updates and coalescing, we rely on a 2-entry ETT (i.e., only two concurrent epochs are allowed, while enforcing the order between them) and the 64-entry PTT is shared by the two epochs. An *sfence* operation is also emulated to demarcate epoch boundaries. For our coalescing out-of-order BMT update scheme, we assume an LCA coalescing where two adjacent updates to the BMT can be coalesced each time, with the leading store stopping at the LCA and delegating the root update to the trailing store.

In our sensitivity study, we vary the latency of the MAC computation (0–80 cycles), epoch size (4–256 stores), metadata cache size (32–256KB), and LLC sizes (1–4MB) to analyze their impacts. Cache latencies are 2 cycles (L1), 20 cycles (L2) and 30 cycles (L3) for their default configurations. The storage required by the PTT (Fig. 7) is as follows: each PTT entry has EID (6 bits) , *V, R* and *P* (3 bits), *Lvl* (4 bits), *WPQptr* (32-bit), and *PendingNode* (32 bits), totalling 77 bits. For 64 entries, the total is 616B. For ETT, each entry has *EID* (6 bits), *V* and *R* (2 bits), *Lvl* (4 bits), and *Start* and *End* (two 6 bits), totalling 24 bits. A 2-entry ETT yields storage

TABLE IV
EVALUATED SCHEMES

Name	Scheme
secure_WB (baseline)	Secure processor scheme with write-back caches and NVMM, which does not support any persistency model
unordered	Write-through metadata and data caches without invariant 2 (BMT root update ordering) enforced, similar to [4]
sp	Strict persistency with sequential updates of BMT
pipeline	Strict persistency with pipelined updates of BMT
o3	Epoch persistency with out-of-order updates of BMT within an epoch, but in order across epochs
coalescing	o3 plus coalescing updates of BMT

overheads of 48 bits.

Benchmarks We use 15 representative benchmarks from SPEC2006 [21] to evaluate the proposed BMT write update models: astar, bwaves, cactusADM, gamess, gcc, gobmk, gromacs, h264ref, leslie3d, milc, namd, povray, sphinx3, tonto, and zeusmp. All benchmarks are fast forwarded to representative regions and run with 100M instructions.

Evaluated Schemes The schemes we used for evaluation are shown in Table IV. For each scheme, we try two configurations. The first one is full memory protection, indicated with suffix '**_full**', where the entire memory is assumed to be persistent and is protected. This is likely to be too pessimistic, because even in persistent memory applications, not all data needs to be persistent and supports crash recovery. The stack, for example, is only used for function parameters, local variables, and spills and refills of registers (especially acute in x86 ISA that has a limited number of general purpose registers), hence it is likely that it only needs memory encryption and integrity verification but without persistency support. Considering these factors, our default evaluation assumes the stack is not persistent, and covers only the heap and static/global region.

VII. EVALUATION RESULTS

Summary As expected, our best performing results come from OOO BMT updates with coalescing (*coalescing*), followed by OOO BMT updates without coalescing (*o3*), pipelined BMT updates (*pipeline*), and finally strict in order BMT updates (*sp*). The overheads compared to the baseline without any persistency (*secure_WB*) for all the schemes are: 720% (*sp*), 210% (*pipeline*), 20.7% (*o3*), and 20.2% (*coalescing*). Our best scheme reduces the overhead by $36\times$ compared to the worst scheme, when protecting the entire memory minus the stack segment. Now, we analyze the performance in more details, starting from strict persistency to epoch persistency, followed by analyzing the performance overheads varying key design parameters.

Strict Persistency Here we compare results from sequential and pipelined BMT updates for strict persistency model, both for full memory protection as well as excluding the stack segment persistency (default).

Fig. 8 shows the execution time of strict persistency (*sp*), pipelined BMT updates (*pipeline*), and strict persistency with unenforced Invariant 2 (BMT updates ordering), similar to [4] (*unordered*) normalized to the *secure_WB* scheme where no persistency is utilized, i.e. no cache line flushes or persist barriers. We can make two observations. First, over the base of no persistency, SP incurs very high performance overheads, incurring a geometric average of $7.2\times$ ($30.7\times$ for full memory). The majority of the overheads comes from the ordered BMT root updates. SP with unordered root updates significantly reduces such performance overhead but does not guarantee BMT verification success on crash recovery. Second, by pipelining BMT updates at different tree levels between persists, our *pipeline* scheme reduces the performance overhead of SP to $2.1\times$ ($6.9\times$ for full-memory), representing speedup ratios of 3.4 (4.4 for full memory).

The key reason for high SP overheads is the high cost of each persist: each store must completely persist all crash recovery tuple, including updating the BMT root. With a MAC computation of 40 cycles and 9 BMT levels, it takes 360 cycles to update the BMT root. Applications that have high rate of stores perform worse than others.

Table V shows the number of persists in different schemes. In *sp_full* and *sp*, the number of persists is the number of all stores and non-stack stores, respectively. For *secure_WB*, the number of persists is the number of writebacks from the LLC. We can see that by persisting all stores non-stack, the *persists per kilo instructions* (PPKI) increase by more than two orders of magnitude (1.61 to 119.51, or to 32.6 for non-stack stores). Combined with the sequential leaf-to-root BMT updates, BMT updates become the dominant performance bottleneck. For example, with *gamess* having non-stack PPKI is 51.38 and 360 cycles to update BMT form leaf to root, we can estimate its IPC (instruction per cycle) as $\frac{1000}{360\times 52} = 0.053$, which is very close to the actual IPC of 0.054. Since its IPC with secure_WB is 2.45, the slowdown is $\frac{2.45}{0.053} = 45.3\times$, matching that shown in Fig. 8. For most benchmarks, the slowdown from *sp* correlates very well with the PPKI. Some benchmarks, such as *leslie3d* and *bwaves*, have high PPKIs but relatively much lower overheads than *gamess*. The reason is that their secure_WB model IPCs are low, due to the high number of dirty-block evictions from LLC.

To better understand the impact of MAC latency, in the next experiment, we vary the MAC latency from 0, 20, 40 to 80 cycles. We also simulate ideal meta-data caches (MDC) that can cache unlimited counters, MACs, and BMT nodes, never miss, and have a zero-cycle MAC computation latency. The results are shown in Fig. 9. From the figure, we can confirm that MAC computation is the key bottleneck of SP. MDC shows negligible performance overheads relatively, pointing out that persisting data and meta-data do not incur much overheads, as long as the MAC latencies involved in BMT

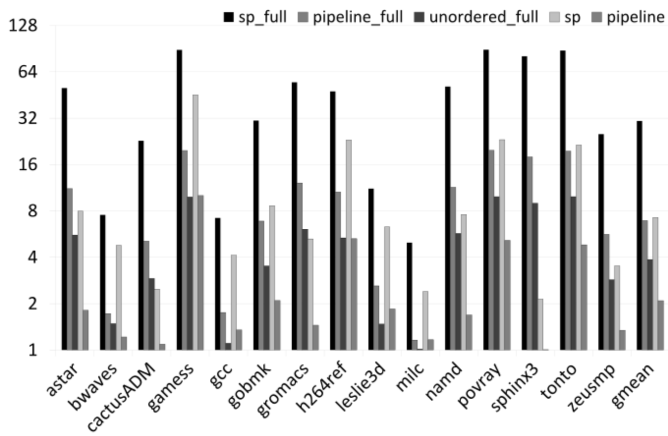


Fig. 8. Execution time of SP schemes normalized to secure_WB model. Scale is log_2.

TABLE V
THE NUMBER OF *persists per kilo instructions* (PPKI). THE NUMBERS IN 'SP_FULL' AND SECURE_WB_FULL' INCLUDE ALL STORES WHILE FOR OTHERS ONLY NON-STACK STORES.

Benchmark	sp_full (num stores)	secure_WB _full (write backs)	sp (num stores)	o3 (epoch stores)
astar	83.48	0.35	13.21	1.97
bwaves	100.27	8.70	61.60	26.47
cactusADM	114.59	1.55	12.35	5.68
gamess	100.72	0	51.38	30.433
gcc	126.73	1.46	67.38	36.64
gobmk	125.16	0.17	34.41	14.63
gromacs	105.73	0.04	9.66	2.69
h264ref	101.17	0	48.80	10.45
leslie3d	108.79	7.78	58.47	17.58
milc	40.18	2	13.65	4.10
namd	133.10	0.18	19.66	2.07
povray	150.72	0	39.23	11.22
sphinx3	184.29	0.10	4.87	1.04
tonto	141.84	0	34.45	16.60
zeusmp	175.87	1.92	19.87	4.66
Average	119.51	1.61	32.60	12.41

leaf-to-root updates incur no cost.

Epoch Persistency We will now discuss results for epoch persistency model, shown in Fig. 10 (y-axes shown in linear scale). Two optimizations are enabled in this model: out of order BMT updates (*o3*) and coalescing BMT updates (*coalescing*). The figure shows *o3* and *coalescing* achieve very low performance overheads: 20.7% and 20.2%, respectively (2.42× and 2.35× for full memory, respectively), compared to the 720% with *sp*. The performance improvements come from two major sources: the overlapping of BMT updates, which reduces the critical path of BMT updates within an epoch, and the large reduction of persists when stores within an epoch fall into the same cache block. The latter can be seen in Table V in the last column. Compared to SP,*o3*'s PPKI is roughly one third of *SP*'s PPKI (12.41 vs. 32.6).

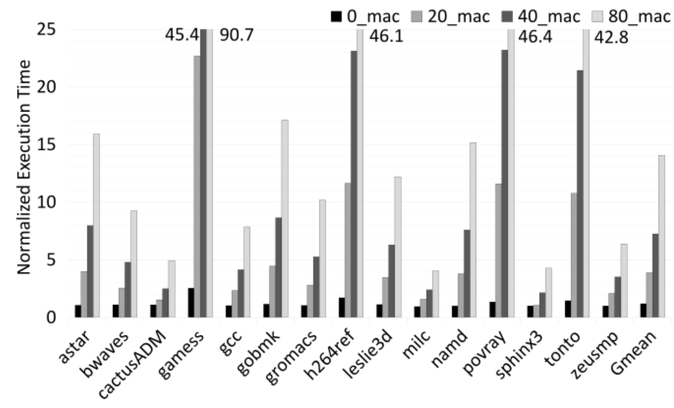


Fig. 9. Execution time of SP normalized to secure_WB with different MAC latencies and ideal metadata caches.

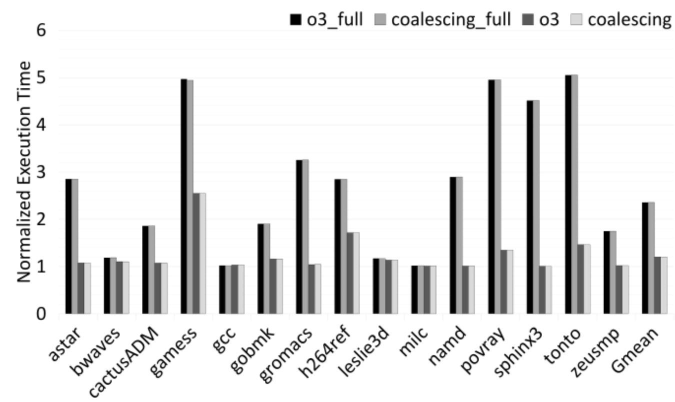


Fig. 10. Execution time of epoch persistency schemes normalized to secure_WB model.

Fig. 10 also shows that *coalescing* has limited impact on performance over *o3*. The reason is that in order to coalesce updates, the older update would wait for the younger one to reach the LCA. Therefore, the saving that comes from coalescing is mainly due to the reduction of the number of updates to BMT nodes. Indeed, our experiments show that coalescing reduces BMT updates by 26.1% on average.

Another interesting observation from Fig. 10 is that in some cases (e.g. *milc*), our optimized epoch persistency model can match or even outperform *secure_WB*. Digging deeper, the reason is that with *secure_WB*, evicted dirty blocks perform BMT updates sequentially rather than pipelined or overlapped in our schemes.

Impact of Epoch Size Fig. 11 shows the impact of epoch size (in number of stores) in affecting persists per kilo instructions (PPKI). As expected, the larger the epoch, the more likely stores within a single epoch to fall into a single cache block that result in fewer persists, as the block is buffered in the cache until the end of the epoch before it is written back to main memory. Thus, naturally we would expect that the performance overheads of our scheme to monotonically decrease with the epoch size. This is true in general, but only up to some point, after which the opposite is observed. Fig. 12 shows the execution time of *coalescing* with varying the epoch size, normalized to *secure_WB*. Upon deeper analysis, we found that while large epochs enable larger reduction in PPKI, small epochs smooth the write traffic to memory [28] hence reducing the queueing delay of persists in the MC and memory. This effect causes an epoch size of 256 to perform worse than 128 for some benchmarks (such as gamess, milc, and zeusmp).

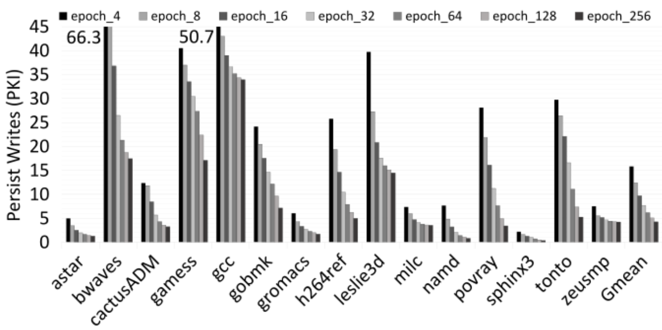


Fig. 11. The number of persists per kilo instruction (PPKI) for different epoch sizes.

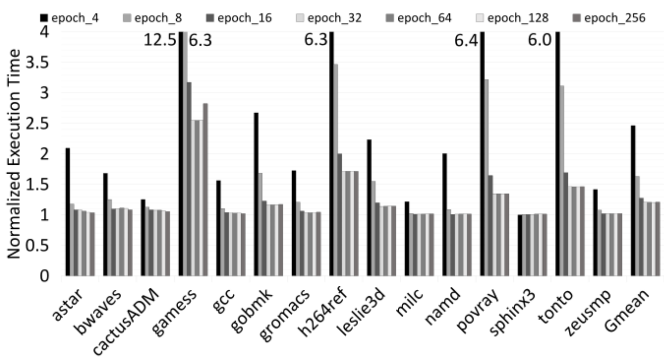


Fig. 12. Execution time of our *coalescing* scheme with different epoch sizes, normalized to secure_WB.

Impact of Write Pending Queue Size In our design, each entry in the WPQ holds a memory update (i.e., a store) until its entire memory tuple is ready to be persisted and the ordering requirement is met. As each store needs to update the BMT, the WPQ size determines how many BMT updates can be overlapped. With the strict persistency model, pipelined BMT overlaps up to nine BMT updates since the BMT has nine levels. Therefore, a WPQ with 9 entries is sufficient. For epoch persistency model, our coalescing BMT schemes allows all stores in an epoch to update the BMT. Therefore, the WPQ size should correspond to the epoch size. We varied the WPQ size from 4 to 64 entries for our *coalescing* BMT model. WPQ sizes below 32 entries displayed increasing overhead, with a WPQ size of 4 showing 12% performance overhead compared to 32 entries. Fewer than 32 WPQ entries reduces performance by limiting the concurrency of BMT updates, but larger than 32 WPQ entries do not add performance improvement over 32 entries. Therefore, we use 32 as our default WPQ size.

Impact of Metadata Cache and LLC Capacity In this experiment, we vary all three metadata caches capacity from 32KB to 256KB. Our results indicate up to 2% performance difference across various sizes for any of our scheme.

We also vary the LLC capacity, from 4MB to 1MB.Our results indicate the performance overheads of coalescing BMT only vary modestly, from 20.2% to 22.8%, when the LLC capacity varies.

VIII. Conclusions

Memory integrity verification and encryption are essential for implementing secure computing systems. Atomically persisting integrity tree roots is responsible for the majority of the overhead incurred by updating security metadata. In this work, we presented three optimizations for atomically persisting NVM Bonsai Merkle Tree roots. With a strict persistency model, our proposed pipelined update mechanism showed an $3.4\times$ performance improvement compared to sequential updates. With the epoch persistency model, our out-of-order root update and update coalescing mechanisms showed performance improvements of $5.99\times$ over sequential updates. These optimizations significantly reduce the time required to update integrity tree roots and pave the way to make secure NVMM practical.

References

[1] M. Alshboul, J. Tuck, and Y. Solihin, “Lazy persistency: A high-performing and write-efficient software persistency technique,” in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[2] T. Aura, “Strategies against replay attacks,” in *Proceedings 10th Computer Security Foundations Workshop*, 1997.

[3] A. Awad, S. Suboh, M. Ye, K. Abu Zubair, and M. Al-Wadi, “Persistently-secure processors: Challenges and opportunities for securing non-volatile memories,” in *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2019.

[4] A. Awad, L. Njilla, and M. Ye, “Triad-nvm: Persistent-security for integrity-protected and encrypted non-volatile memories (nvms),” in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” *SIGARCH Comput. Archit. News*, 2011.

[6] G. E. Blelloch, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun, “The parallel persistent memory model,” in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, 2018.

[7] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, “Atlas: Leveraging locks for non-volatile memory consistency,” in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014.

[8] S. Chhabra and Y. Solihin, “i-nvmm: A secure non-volatile main memory system with incremental encryption,” in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011.

[9] S. Chhabra, B. Rogers, and Y. Solihin, “Shieldstrap: Making secure processors truly secure,” in *Proceedings of the 2009 IEEE International Conference on Computer Design*, 2009.

[10] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, “Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories,” in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[11] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, “Better i/o through byte-addressable, persistent memory,” in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

[12] V. Costan and S. Devadas, “Intel sgx explained,” Cryptology ePrint Archive, Report 2016/086, 2016.

[13] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014.

[14] H. Elnawawy, M. Alshboul, J. Tuck, and Y. Solihin, "Efficient check-pointing of loop-based codes for non-volatile main memory," in *26th International Conference on Parallel Architectures and Compilation Techniques*, 2017.

[15] C. Fletcher, M. van Dijk, and S. Devadas, "A secure processor architecture for encrypted computation on untrusted programs," in *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing*, 2012.

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

[17] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, "Persistency for synchronization-free regions," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018.

[18] S. Gueron, "A memory encryption engine suitable for general purpose processors," 2016.

[19] L. Guo, Y. Zhang, and F. X. Lin, "Let the cloud watch over your iot file systems," *CoRR*, vol. abs/1902.06327, 2019. [Online]. Available: http://arxiv.org/abs/1902.06327

[20] J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. Felten, "Lest we remember: Cold boot attacks on encryption keys," in *USENIX Security Symposium*, 2008.

[21] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, 2006.

[22] Intel, "Intel and Micron produce breakthrough memory technology," 2015.

[23] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via justdo logging," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[24] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "Atom: Atomic durability in non-volatile memory through hardware logging," in *2017 IEEE International Symposium on High Performance Computer Architecture*, 2017.

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

[26] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language-level persistency," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

[27] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-performance transactions for persistent memories," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[28] H. S. Lee, G. S. Tyson, and M. K. Farrens, "Eager writeback - a technique for improving bandwidth utilization," in *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 33, Monterey, California, USA, December 10-13, 2000*, 2000, pp. 11–21. [Online]. Available: https://doi.org/10.1109/MICRO.2000.898054

[29] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Zhenghong Wang, "Architecture for protecting critical secrets in microprocessors," in *32nd International Symposium on Computer Architecture (ISCA'05)*, 2005.

[30] T. S. Lehman, A. D. Hilton, and B. C. Lee, "Poisonivy: Safe speculation for secure memory," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016.

[31] T. S. Lehman, A. D. Hilton, and B. C. Lee, "Maps: Understanding metadata access patterns in secure memory," *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 33–43, 2018.

[32] C. Liu and C. Yang, "Secure and durable (sedura): An integrated encryption and wear-leveling framework for pcm-based main memory," in *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2015 CD-ROM*, 2015.

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

[34] S. Liu, K. Seemakhupt, G. Pekhimenko, A. Kolli, and S. Khan, "Janus: Optimizing memory and storage support for non-volatile memory systems," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19, 2019.

[35] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, 2014.

[36] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[37] X. Pan, A. Bacha, S. Rudolph, L. Zhou, Y. Zhang, and R. Teodorescu, "Nvcool: When non-volatile caches meet cold boot attacks," *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 2018.

[38] S. Pelley, P. Chen, and T. Wenisch, "Memory persistency," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, 2014.

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

[40] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "Thynvm: Enabling software-transparent crash consistency in persistent memory systems," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015.

[41] P. Roberts, "Mit: Discarded hard drives yield private info," *ComputerWorld*, 2003.

[42] B. Rogers, , M. Prvulovic, and Y. Solihin, "Effective data protection for distributed shared memory multiprocessors," in *in Proceedings of the International Conference of Parallel Architecture and Compilation Techniques (PACT*, 2006.

[43] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "Using address independent seed encryption and bonsai merkle trees to make secure processors os- and performance-friendly," *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007.

[44] B. Rogers, C. Yan, S. Chhabra, M. Prvulovic, and Y. Solihin, "Single-level integrity and confidentiality protection for distributed shared memory multiprocessors," in *in Proceedings of the 14th International Symposium on High Performance Computer Architecture (HPCA-14*, 2008.

[45] A. Rudoff, "Deprecating the pcommit instruction," 2016.

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

[47] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, J. Joao, and M. Qureshi, "Morphable counters: Enabling compact integrity trees for low-overhead secure memories," *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[48] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for nvm," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.

[49] S. Shin, J. Tuck, and Y. Solihin, "Hiding the long latency of persist barriers using speculative execution," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

[50] G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, 2003.

[51] G. E. Suh, C. W. O'Donnell, and S. Devadas, "Aegis: A single-chip secure processor," *IEEE Design Test of Computers*, 2007.

[52] G. Suh, D. E. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *in Proceedings of the International Symposium on Microarchitecture (MICRO*, 2003.

[53] S. Swami and K. Mohanram, "Acme: Advanced counter mode encryption for secure non-volatile memories," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018.

[54] S. Swami, J. Rakshit, and K. Mohanram, "Stash: Security architecture for smart hybrid memories," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018.

[55] S. Swami and K. Mohanram, “Arsenal: Architecture for secure non-volatile memories,” *Computer Architecture Letters*, 2018.
[56] M. Taassori, A. Shafiee, and R. Balasubramonian, “Vault: Reducing paging overheads in sgx with efficient integrity verification structures,” in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.
[57] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, “Architectural support for copy and tamper resistant software,” in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.
[58] J. Vijayan, “Data breaches probed at new jersey blue cross, georgetown,” *ComputerWorld*, 2011.
[59] H. Volos, A. J. Tack, and M. M. Swift, “Mnemosyne: Lightweight persistent memory,” in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.
[60] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, “Improving cost, performance, and security of memory encryption and authentication,” in *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*, 2006.
[61] J. Yang, Y. Zhang, and L. Gao, “Fast secure processor for inhibiting software piracy and tampering,” in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.
[62] M. Ye, C. Huges, and A. Awad, “Osiris: A low-cost mechanism to enable restoration of secure non-volatile memories,” 2018.
[63] L. Zhang and S. Swanson, “Pangolin: A fault-tolerant persistent memory programming library,” in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, 2019.
[64] Y. Zhang, L. Gao, J. Yang, and R. Gupta, “Senss: Security enhancement to symmetric shared memory multiprocessors,” in *in Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2005.
[65] Y. Zou and M. Lin, “Fast: A frequency-aware skewed merkle tree for fpga-secured embedded systems,” in *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2019.
[66] K. A. Zubair and A. Awad, “Anubis: Ultra-low overhead and recovery time for secure non-volatile memories,” in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019.
[67] P. Zuo and Y. Hua, “Secpm: a secure and persistent memory system for non-volatile memory,” in *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.
[68] P. Zuo, Y. Hua, and Y. Xie, “Supermem: Enabling application-transparent secure persistent memory with low overheads,” in *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.