EAI Endorsed Transactions
on Security and Safety

Research Article EAI.EU

Caching Techniques for Security Metadata in Integrity-Protected Fabric-Attached Memories

Mazen Alwadi*, Amro Awad

University of Central Florida, Electrical and Computer Engineering Department, Orlando, FL, United States

Abstract

The constant need for larger memories and the diversity of workloads have drove the system vendors away from the conventional processor-centric architecture into a memory-centric architecture. Memory-centric architecture, allows multiple computing nodes to connect to a huge shared memory pool and access it directly. To improve the performance, each node uses a small local memory to cache the data. These architectures introduce several problems when memory encryption and integrity verification are implemented. For instance, using a single integrity tree to protect both memories can introduce unnecessary overheads. Therefore, we propose Split-Tree, which implements a separate integrity tree for each memory. Later, we analyze the system performance, and the security metadata caches behavior when separate trees are used. We use the gathered insights to improve the security metadata caching for the separate trees and ultimately improve the system performance.

Received on 16 June 2020; accepted on 07 July 2020; published on 11 August 2020

Keywords: Fabric-Attached Memory, Secure Memory, Memory-Centric, Encrypted Memory, Integrity Tree

doi:10.4108/eai.13-7-2018.165516

1. Introduction

The continuous growth in large-scale computing, data-centric workloads, and the proliferation of virtual machines points to the need of greater memory capacities. Moreover, with the huge spectrum of workloads diversity and memory requirements, increasing the memory capacity of each computing node can lead to highly under-utilized memories [21]. Therefore, Fabric-Attached Memory (FAM) architecture are becoming more desirable. FAM architectures allows computing nodes to seamlessly connect to a huge memory pool. In FAM architectures, a memory semantic protocol defines how different processing elements can interact with the memory through the system fabric. Gen-Z [3] is a prominent example of FAM protocols, where different processing agents (can be from different vendors) are directly attached to a shared fabric that connects to one or more different memory pools. Each memory pool implements its own media controller that translates Gen-Z commands into media-specific read and write operations. Processing elements must use the Gen-Z command format and standards to be able to access the fabric-attached memory. While Gen-Z is perhaps the de facto standard for FAM architectures, many new protocols advocate for similar directions, e.g., Compute Express Lanes (CXL) [2] and Cache Coherent Interconnect for Accelerators (CCIX) [1]. Such protocols introduce a new architecture where processing elements can access the shared memory pools without going through *home* processor as in conventional system architectures such as NUMA systems. As FAM architectures are expected to have huge memory pools, using DRAM as the main memory can be problematic due to the expensive power required for the frequent refresh operations and cooling. Therefore, FAM architectures are expected to use emerging Non-Volatile-Memories (NVM) as the main memory [3].

Emerging NVMs are promising replacement for DRAM in future computing systems [17, 18, 20]. Such NVMs feature high-density, ultra-low idle power, performance comparable to DRAM and persistency. On the other hand, NVMs have a limited write endurance and power consuming writes. Moreover, NVMs ability to retain data during power loss facilitates data remanance attacks. Therefore, NVMs are typically

*Corresponding author. Email: mazen.alwadi@knights.ucf.edu

coupled with security features i.e, encryption and integrity verification [5, 7, 8, 30, 31, 33, 34]. While the secure memory implementations are not limited to NVMs due to similar data remanance attacks performed in DRAM i.e., cold boot attack [32], our work is not limited to NVMs as well.

Secure memory implementation in state-of-the-art work [7, 11, 28–31, 33, 34] uses counter-mode encryption to protect the data confidentiality and Merkle-Tree for integrity verification. Enabling security metadata can lead to significant performance overheads, as each memory access requires accessing the corresponding encryption counter and possibly the whole Merkle-Tree branch. Reading/Updating a cacheline can lead to tens of Merkle-Tree reads/writes, which grows as the size of the protected memory region increases. Typically, security metadata caches are used to reduce the required memory accesses for encryption/decryption and integrity verification [19]. Previous work in secure memory architecture [7, 28–31, 33] considered a system that has a single memory pool protected by a single Merkle-Tree. However, each node in the FAM architecture has an access to a private local memory, and a global shared memory. While the global memory is shared between computing nodes, the nodes are not necessarily sharing data or the same memory region, as each node can be assigned to different region (i.e., within the same 1 TB). Therefore, using a single Merkle-Tree can introduce unnecessary overheads.

Using a single Merkle-Tree requires the tree to protect the local memories as well as the global memory. Which can lead to updating the Merkle-Tree each time a local memory cacheline is written. Taking the atomicity requirement into consideration, a single cacheline update requires updating the whole Merkle-Tree branch alongside with the updated cacheline. This atomic update process will require locking the whole Merkle-Tree branch until the update is finished, which can prevent other processing elements from accessing the affected Merkle-Tree nodes until the update is finished. Moreover, such approach will enforce the local memory to operate as a write through cache, as each write needs to update the global memory data and its associated security metadata. Thus, Split-Tree implements two different Merkle-Trees.

Having two separate Merkle-Trees improves the system performance and reduces the number of memory accesses required for integrity verification. Since the local memory is significantly smaller than the global shared memory, the Merkle-Tree protecting the local memory is smaller and contains less levels. Moreover, having a separate Merkle-Tree for the local memory eliminates the need to update the global memory Merkle-Tree when a local memory cacheline is written, the global memory Merkle-Tree is only updated when the data is written back to the global memory. On the other hand, having two separate Merkle-Trees can affect the system performance, especially in terms of metadata caching. For instance, having a large Merkle-Tree protecting the nodes' portion of the global memory can generate high demand over the shared security metadata cache. While the global memory has access latencies higher than the local memory, thus caching security metadata related to the global memory can potentially eliminate several requests to verify the integrity of a global memory resident data. On the other hand, the local memory is accessed more frequently than the global memory. Therefore, caching the local memory Merkle-Tree nodes can reduce multiple requests to the local memory, which happens more often than the global memory accesses. In this work, we study the problem of secure memory implementation in FAM architectures, and propose using a Split-Tree scheme to protect the local and global memories. Moreover, we propose a caching scheme that allows caching the global memory security metadata in the local memory, and partitions the security metadata cache to reduce the contention over the security metadata cache.

2. Related Work

Secure memory implementation has been studied from different perspectives by variety of studies. Osiris [30] discussed the crash consistency problem of secure NVMs, and highlighted that a power failure or a crash can result in having stale encryption counters, which can lead to integrity verification failure, and thus losing the whole memory content. Osiris proposed a scheme to recover the encryption counters after a crash, which relies on a stop-loss mechanism coupled with using ECC bits as a sanity check for the recovered counter correctness. Anubis [33] addresses the recovery time problem in secure NVMs. Anubis emphasizes that recovering the encryption counters is not always sufficient to recover the integrity tree. Moreover, rebuilding the integrity tree can take hours for practical size NVMs. Therefore, Anubis proposed a scheme that tracks the updated security metadata cachelines in the cache. During the recovery phase, Anubis relies on the tracking mechanism to pin-point the lost data and recover it. Phoenix [5] highlights the overheads caused by Anubis scheme when a ToC is used. Phoenix [5] aims to reduce the number of writes incurred to recover the ToC by utilizing an encryption counters recovery scheme to recover the encryption counters, and tracks the updates of unrecoverable intermediate ToC nodes. VAULT [29] discussed the overheads caused by the integrity tree and proposed a scheme to reduce these overheads. VAULT proposed having a variable arity tree, in which lower integrity tree levels can pack more child nodes and

the arity decreases as we go higher, until it saturates at an arity of eight. VAULT reduces the depth of the integrity tree and thus reduces the number of accesses required to verify the integrity of an encryption counter or update it. Synergy [28] discussed the overheads of Message Authentication Codes (MACs) associated with the data in secure memory architectures. As counter mode encryption is used to protect the data confidentiality, and Merkle-Tree is used to verify the encryption counters integrity, the data integrity is protected by calculating a MAC value over the data and the encryption counter. Therefore, protecting the encryption counters is sufficient to ensure the integrity of the data, due to the attacker inability to generate the same MAC values [24]. Synergy [28] aims to reduce the number of memory access required for integrity verification by replacing the ECC bits with MAC value, and storing the ECC bits in the memory instead. Synergy relies on the fact that MAC values can be used for error detection as well, and will always be required for integrity verification. On the other hand, ECC bits are used for error checking and rarely used for error correction. Rogers et al.[25] proposed a scheme for data protection in Non-Uniformed Memory Access (NUMA) systems. The proposed scheme assumes each node is protecting its memory, which leaves the interconnects and message communication between nodes unprotected. To protect the interconnects and the communicated messages, a point-point encryption is used. The encrypted messages are associated with MAC values to ensure their integrity. Morphable counters [27] discussed the overheads of secure memory implementation, and suggested that increasing the encryption counters cacheability can increase the cache hit rate and improve the performance. Morphable counters proposes a scheme that allows packing a maximum of 128 encryption counters per cacheline, but reduces the counters size and uses some bits for the management. However, using small counters can cause frequent minor counters overflow which can lead to the whole page being re-encrypted. Therefore, Morphable counters discusses the trade offs between counters cacheability and the overflow.

3. Background

3.1. Threat Model

In this work, we assume a similar threat mode as in state-of-the-art work in secure memory architecture [5, 7, 8, 22, 27, 28, 30, 31, 33]. The trust base is limited to the processor and its internal structures. We assume an attacker who can snoop the local memory bus and the global memory bus, scan the memories content, tamper with memories content, and replay old packets. Differential power attacks, electromagnetic inference attacks, and attacks targeting the processor speculative execution such as Spectre and Meltdown are beyond the scope of this work.

3.2. Emerging Non-Volatile Memories (NVMs)

Emerging Non-Volatile Memories (NVMs) are expected to replace the DRAM as main memories [5, 7, 27–31, 33]. Emerging NVMs combine the features of main memory and storage, as they feature byte addressability, access latencies comparable to DRAM, near-zero idle power consumption, high density, and the ability to retain data during power failure episodes. Data persistency of NVMs is probably the most promising feature as it enables persistent applications such checkpointing and file systems. However, the persistency feature facilitates the data remanance attacks [30]. Therefore, NVMs are typically shipped with confidentiality protection and integrity verification features [33]. However, NVMs suffer from power consuming writes, and limited write endurance. In a matter of fact, the most promising NVM technology, Phase-Changed Memory (PCM), can only endure tens of millions of writes [6]. Adding encryption to NVMs exacerbates the write endurance problem, due to the encryption diffusion property. Moreover, updating a data cacheline in a Merkle-Tree integrity protected system can lead to tens of Merkle-Tree updates. Thus, NVM friendly encryption and integrity verification algorithms are being explored by the research community.

3.3. Counter Mode Encryption

Split counter-mode encryption is used in state-of-the-art implementations of secure memory architecture [7, 22, 27, 30, 31, 33]. Counter mode encryption thwarts dictionary based attacks, bus snooping attacks, and known-plaintext attacks. Additionally, Counter mode encryption does not propagate errors as the input of a stage does not depend on the output of previous stages. Moreover, Counter-Mode encryption overlaps the One-Time-Pad (OTP) generation with memory read latency, thus hides the decryption latency except for the XOR operation. Figure 1 shows the split counter-mode encryption scheme. Split counter-mode encryption assigns a minor counter (7-bit) for each data cacheline, and a major counter (64-bit) for each page. An Initialization Vector (IV) composed of the page ID, page offset, minor counter, major counter, and padding. A secure processor key is used to generate the OTP by encrypting the IV using AES encryption engine. Then, the cacheline is XORed with the OTP to do the encryption/decryption. To ensure the security of counter mode encryption, re-using encryption counters is prohibited as it facilitates known-plaintext attacks [7, 30, 33].

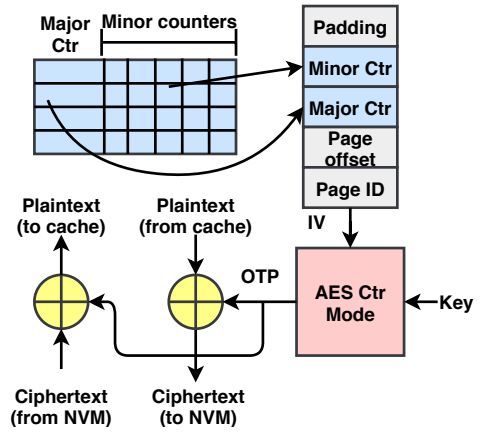

Figure 1. Split-Counter Mode Encryption

3.4. Integrity Trees

Bonsai Merkle-Tree (BMT). While the integrity of the data can be easily verified using a keyed Message Authentication Code (HMAC) values calculated over the data and the encryption counters [11, 24], it would be sufficient to protect the encryption counters using a hash tree with its' root kept secure in the processor. The BMT is a tree of hashes built on top of the encryption counters to ensure the integrity of the encryption counters. The BMT calculates the hash of encryption counters as shown in Fig.2 to create the first level of the tree. Then, it calculates the hashes of first level nodes to generate the second level and so on. The processes of hashing is continued recursively until a single node is calculated, which is referred to as the root. The BMT calculates the hash of 64 encryption counters to generate the first level, and hashes each 8 nodes (arity of eight) to form upper levels. Figure 2 shows a BMT with arity of two.

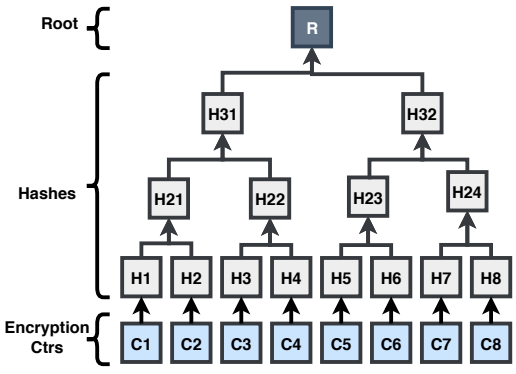

Figure 2. Bonsai Merkle-Tree

Whenever an encryption counter is fetched from the memory, its' integrity needs to be verified by calculating the hashes until a root is generated. If the calculated root matches the processor stored root, the encryption counter integrity is verified. Similarly, when a dirty encryption counter is written to the memory, the whole BMT branch needs to be updated. A faster way to verify the counter integrity can be achieved by stopping the verification process with the first parent cache hit, as the cached nodes' integrity was verified when it was brought to the processor cache.

Tree of Counters (ToC). The ToC shown in Fig.3 is a parallelizable form of the MT. The ToC uses 56-bit counter for each data cacheline, and packs each eight counters (arity of eight) along with a 56-bit MAC value, and an unused eight bits in a single cacheline [11, 33]. The nodes' MAC value is calculated over the node counters along with a counter from the parent node. The integrity verification in the ToC is similar to the BMT, but the update process is different. Whenever an encryption counter is updated, the corresponding counter in the parent node is incremented and the MAC value is updated. Since the upper nodes update does not depend on the update of the child nodes, the update process can be done in parallel [11, 33].

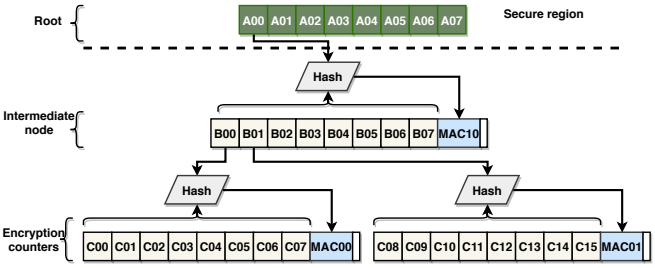

Figure 3. Tree of Counters.

Integrity Tree Update Schemes. Both BMT and ToC can be either eagerly or lazily updated [7, 30, 31, 33]. An eager update scheme ensures the root always reflects the most recent state of the memory, which requires updating the whole MT branch with the root for each encryption counter update. Such an update scheme incurs multiple memory accesses for each update, and significantly degrades the performance. Despite the performance degradation, it allows recovery after a crash in case of recovery expectation (NVM main memory) if all the updates were persisted. In case of the MT were eagerly updated but the updates were not strictly persisted, the BMT can be rebuilt if only the encryption counters are persisted, but the nodes inter-dependencies of the ToC makes it impossible to recover from the encryption counters [5, 33].

The lazy-update scheme updates only the encryption counter and relies on the natural eviction to propagate the updates upwardly [5, 7, 33]. Whenever a dirty node is evicted, its' parent is fetched and updated. Such an update scheme reduces the memory accesses and the performance overheads significantly, but will have a stale root value, and relies on the cached security

metadata to represent the most recent state of the memory. Thereby, systems implementing a lazy-update scheme are performance friendly, but more susceptible to crash consistency problems [7, 22, 30].

3.5. Fabric-Attached Memory (FAM)

FAM architecture differs from traditional processor centric architecture by disaggregating the memory from the processing unit, and implements a shared large memory pool. The memory pool can be accessed directly by any processing element without the need to go through a home processor as in NUMA systems [16]. The main enabler of the FAM architectures are FAM protocols such as Gen-Z [3], CCIX [1], and CXL [2]. FAM protocols which are being currently developed by the joint efforts of leading system providers such as Google, HP, IBM, Dell EMC, Micron. Such protocols, requires the processing elements to implement the memory semantic protocol at the memory controller to access the shared memory pool [3].

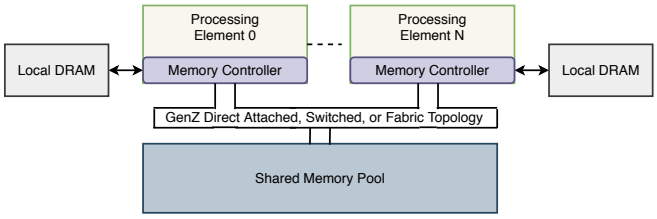

Figure 4. Fabric-Attached Memory Architecture.

FAM architecture, shown in Figure 4, has several advantages over traditional NUMA systems. For instance, for a processing element to access data in a different nodes' memory, the request has to go through the home processor of that specific memory, which is typically done using expensive message passing protocols. In FAM architecture, sharing data does not require moving the data from one node to another, or sending a request to home processor, data sharing is a matter of assigning the memory region to the requester.

3.6. Motivation

In FAM architecture, each node has an access to a local DRAM and an access to a global shared memory pool. The local DRAM is used to cache the global memory data to improve the system performance. Therefore, implementing secure memory requires special handling, due to having two separate memories. Implementing integrity verification using a single Merkle-Tree as used in traditional systems can incur high overheads. Moreover, a single Merkle-Tree covering the global shared memory and the private local memories of all the nodes in the system can significantly reduce the global NVM lifetime. Additionally, as the Merkle-Tree nodes are required to be updated atomically with the data, this atomicity requirement can lead to even higher overheads, as writing a cacheline at any nodes' local memory will require locking the Merkle-Tree branch covering the modified cacheline until the atomic Merkle-Tree branch update is performed. Moreover, using a single Merkle-Tree will require persisting the whole Merkle-Tree branch and the updated node atomically, which will cause the local DRAM to operate in a write-through manner.

To reduce the overheads of secure memory implementations, a Split-Tree scheme can be used. In the Split-Tree scheme, a Merkle-Tree is used for the global memory, and another Merkle-Tree is used to protect the local DRAM. However, using two different trees can introduce caching problems as the security metadata for the trees will be contesting over the cache resources. Thus, caching the security metadata for both memories can cause a contention over the cache resources and lead to unnecessary overheads. For instance, the global memory is only accessed when the required data are not present in the local memory, but the global memory Merkle-Tree is expected to be huge. On the other hand, the local memory which is frequently accessed have smaller Merkle-Tree. However, the access latencies for the global memory is much higher than the access latencies for the local memory, and for the local memory security metadata to contest the cache resources with the global memory can degrade the system performance. Moreover, security metadata caching can have drastically different behavior depending on the used Merkle-Tree update scheme. As using an eager-update scheme will cause Merkle-Tree nodes in higher levels to be always cashed, as they will be used more frequently. Once the global memory is put into perspective, global memory security metadata cachelines are highly likely to be evicted due to the less frequent accesses.

On the other hand, using a lazy-update scheme tend to cache the lowest levels of the Merkle-Tree and the encryption counters, as it does not require to update the whole tree path for each access. However, using a lazy-update scheme can lead to crash consistency problems in systems with recovery expectations. As FAM architectures are expected to use NVMs as the shared memory pool, lazy-update scheme is not suitable for such systems. To address these challenges, we propose Split-Tree, a scheme that uses separate integirty trees to secure FAM architecture, and provides a security metadata cache partitioning scheme that studies the trade-offs and performance overheads of caching security metadata in FAM architectures.

4. Design

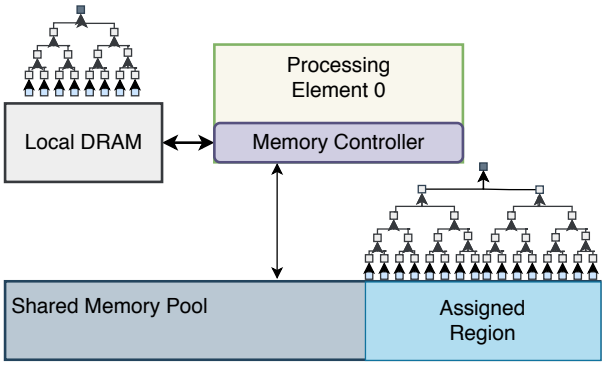

Figure 5. Split Merkle-Trees Design.

4.1. Overview

In secure FAM architecture each processing element is expected to have a memory region of the shared global memory as well as a small local memory. Therefore, each processing element is responsible for implementing data confidentiality and integrity support for its associated memories. Since the local memory is used to cache the global region data, a naive approach would be to apply the encryption and integrity verification for the global memory, and use the same encryption counter and integrity tree to verify the integrity of the local memory data. However, the size of the security metadata responsible for protecting such large memory region is expected to be huge. For instance, to protect a memory region of 1TB using split-counter mode encryption and BMT, it would require 16GB for encryption counters and 11 levels BMT of about 19GB. Having an integrity tree with this huge size will make integrity verification a very costly process, even when caching is used. Therefore, we propose a design that has two separate Merkle-Trees, a large one used to protect the integrity of the owned global memory region, and a smaller one protecting the local memory. We refer to the shared memory region Merkle-Tree as global MT, and to the local memory Merkle-Tree as the local MT.

Figure 5 shows the implementation of the two Merkle-Trees. The Split-Tree scheme has several advantages over the single Merkle-Tree scheme. For instance, the shared memory region should be encrypted using different encryption counters and different processor key to allow data sharing, and preserve each processing element private data. As if the shared region needs to be re-assigned to a different processing element, the new global region owner needs to posses the used encryption key, have access to the most recent encryption counters, and the most recent integrity tree root. Using a single Merkle-Tree means all the processing elements in the system should have the same encryption key. In case of a malicious processing element, or if an attacker obtains access to one processing element, the attacker can access any data in the system that belongs to different nodes. Therefore, using the same encryption key will compromise the security of the other processing elements in the system. Moreover, as the encryption keys are different, the Merkle-Tree protecting the global region is different from the one protecting the local memory to enable data sharing while preserving the security of each processing element. Data sharing can be enabled by passing the region encryption key to the requester along with the Merkle-Tree root, which can be done securely as described in earlier work [25, 26].

Additionally, resolving any security metadata cache miss will require accessing the global memory. However, the global memory access latencies are expected to be 300ns for reads and 1000ns for writes [16]. On the other hand, having a Local MT can obtain the security metadata from the local DRAM.

Moreover, having two different Merkle-Trees can reduce the writes to the NVM global memory by updating the local MT when a data cacheline is evicted from the processor caches to the DRAM. Thus, increases the NVM lifetime which has a limited write endurance. However, having two different Merkle-Trees and two different set of encryption counters requires decrypting/re-encrypting the data when transferred between the different memories.

4.2. Data Transfer Between Memories

Each node in FAM architecture is expected to have a local memory used to cache the data from the assigned global memory region. For performance reasons, the local memory is expected to be a DRAM, but the global memory is expected to be a NVM for higher capacity, persistency and lower power requirements. As the DRAM is used to cache the global memory region, data transfer between the local memory and the global memory is expected to be managed by an extension of the memory controller as in Intel Xeon scalable processor memory controller for Intel Optane DC-Memory mode [4]. Having a single Merkle-Tree to protect the memory integrity will require the encrypted data to be migrated from the global memory to the local memory, and then perform the decryption when the data is fetched from the local memory to the processor cache hierarchy. While this scheme can simplify the implementation of security measures, it requires fetching the whole Merkle-Tree branch to verify the integrity of the required data. Moreover, updating the Merkle-Tree can be expensive in terms of performance overhead, extra writes to the global memory, and NVM lifetime.

On the other hand, using split Merkle-Trees requires decrypting the data as it gets migrated from the global memory, and re-encrypting it as it gets inserted into the

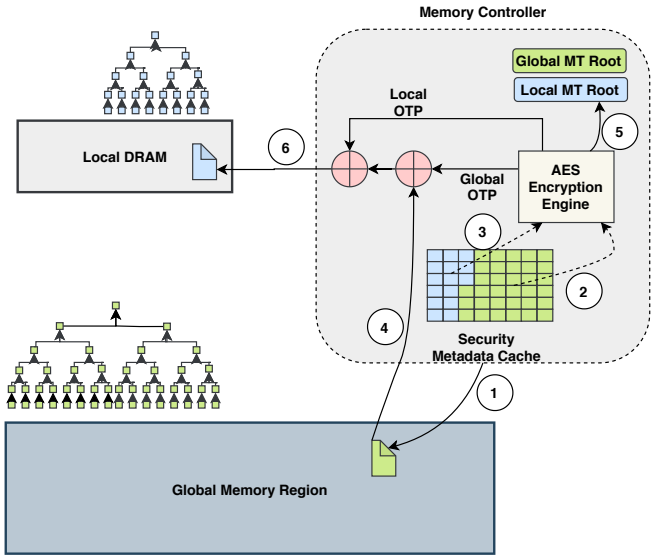

Figure 6. Split Merkle-Trees re-Encryption.

DRAM, which can be done at the memory controller responsible for page migration.

Figure 6 shows how the re-encryption process is performed when a page is migrated from the global memory region to the local memory. In Step 1, the memory controller initiates a page migration from the global memory to the local memory, which requires allocating a physical frame in the local memory for the requested page. While the page is being fetched from the global memory, the memory controller generates the global memory page OTPs and the allocated local memory OTPs by notifying the AES encryption engine as shown in steps 2 and 3. When the data arrives from the global memory, the encrypted data is XORed with the global memory OTP to complete the decryption. After that, the decrypted data is XORed with the local memory OTP to complete the local memory encryption as shown in Step 4. After the re-encryption is done, the local MT root is updated in Step 5. Finally, the data is sent to the local memory in Step 6. Note that generating the OTPs in a timely manner requires the security metadata to be cached. However, caching security metadata can be problematic as the global MT is huge, yet the accesses to the global memory are less frequent than the local memory. Therefore, the local MT nodes will be replacing the global MT nodes in the cache due to the LRU replacement policy.

4.3. Caching Security Metadata

Due to the frequent accesses to the local memory, the security metadata of the global MT will be evicted from the security metadata cache, resulting in multiple accesses to verify the integrity for the global memory to resolve a miss. However, as the local memory is

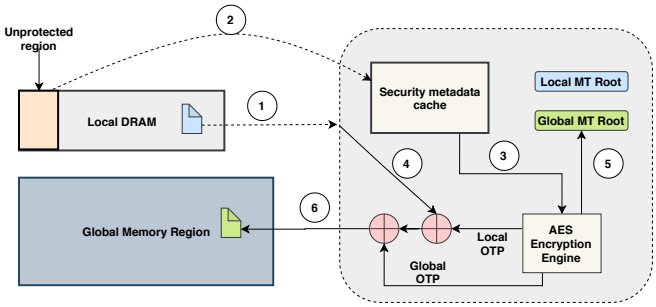

Figure 7. Updating Global Memory Region Merkle-Tree.

a DRAM, it does not require a recovery mechanism. Therefore, using a lazy update scheme to update the local memory Merkle-Tree can reduce the number of accessed metadata for each update. On the other hand, as the global memory region is a NVM, the system should be able to verify the integrity of the NVM data after crashes. Therefore, we use an eager update scheme to update the global memory. Note that, data is written to the global memory region when it gets evicted from the local DRAM.

While using a lazy-update scheme to update the local Merkle-Tree can prevent the local MT nodes from evicting the global MT nodes, but it may lead to the global MT nodes evicting the local MT nodes due to the eager update scheme used for the global MT. Therefore, we partition the security metadata cache to prevent premature eviction of the security metadata cachelines. While partitioning the cache can prevent evicting the security metadata of the global MT, it would still have to access the global memory to fetch the required metadata. To reduce the global memory region accesses, we use a small unprotected region in the local memory to cache the security metadata of the global memory region. Thus, whenever a global memory security metadata cacheline is evicted from the cache, the block is written back to the unprotected memory region as well as the global memory region. Caching the global memory region security metadata in the DRAM reduces the access time of the required metadata to a DRAM access latency instead of the global FAM region. Figure 7 shows the global MT is updated when a dirty page is written back from the DRAM to the global memory region. In Step 1, the memory controller selects a page in the DRAM to be replaced. After that, the memory controller reads the security metadata of the page from the unprotected region in the DRAM and uses the security metadata to generate the OTPs as in steps 2 and 3. In Step 4, the data is decrypted using the local memory OTP and re-encrypted using the global memory OTP. Finally, the global MT root is updated and the data is written back to the global memory region in steps 5 and 6. Note that the global memory region security metadata are only

updated when the new global OTP is generated, which requires updating the whole global MT branch and the root. Finally, writing the data back to the global memory region (NVM) should be done in an atomic manner, in which the data is written atomically along with its associated security metadata. Write atomicity can be achieved by utilizing the Write Pending Queue (WPQ), which is a persistent buffer in the memory controller. The WPQ is supported by the Asynchronous DRAM Self-Refresh (ADR) which provides enough power to flush the WPQ contents in case of a crash.

4.4. Design Discussion

In this section, we discuss some design options and the overheads of our scheme.

The overhead of the re-encryption process is minimal as counter-mode encryption scheme is used, the OTPs generation time can be overlapped with memory read time, and XORing the encrypted data with the OTP completes the decryption. Therefore, the overhead of the re-encryption process is limited to few cycles required to perform the XOR operations.

Both BMT and ToC can be used to provide the required integrity verification. However, in our scheme we use a BMT to protect the integrity of the local memory and a ToC for the global memory. The BMT does not allow a parallel update, but the leaves of the BMT has a higher coverage than ToC leaves, as each leaf node in the BMT covers one page (4KB) of data. On the other hand, the global memory region requires recoverability measures and thus an eager update scheme is required. We use a ToC due to the performance advantage by allowing parallel updates.

As the security metadata of the global memory region is cached in the local DRAM, the DRAM region used for caching is unprotected for two reasons. First, protecting the caching region will require updating the local memory MT and encryption counters each time a cacheline is written, which can introduce unnecessary overheads. Second, the encryption counters are already protected using the Merkle-Tree, which has its root stored securely in the processor.

4.5. Security Discussion

The security of the memories is protected using the encryption counters and the Merkle-Trees. As the re-encryption process is performed inside the secure region, our scheme does not affect the security of the system. However, to ensure the security of the counter-mode encryption scheme, encryption counters re-use is prohibited. Therefore, in a post-crash situation the encryption counters for the local memory (DRAM) are reset, which can open a room for encryption counters reuse. Thus, the encryption key used to encrypt the local memory data should be changed after each crash.

Table 1. Configurations of the Simulated System.

Processing Element (PE)	
Processor	4 Cores, X86-64, Out-of-Order, 2.00GHz
L1 Cache	Private, 4 Cycles, 32KB,8-Way
L2 Cache	Private, 6 Cycles, 256KB, 8-Way
L3 Cache	Shared, 12 Cycles, 1MB/core, 16-Way
Cacheline Size	64Byte
Fabric latency	40 ns
Memory	
Local Memory	256MB DRAM
Global Memory Region Capacity	16GB
NVM Latencies	Read 300ns, Write 1000ns
Encryption Parameters	
Security Metadata Cache	256KB, 8-Way, 64B Block

On the other hand, changing the encryption key for the global memory region (NVM) is not required as the encryption counters are persisted.

Note that passive attacks aiming to read the data are not possible, as the data is encrypted in the memories and the bus. Active attacks trying to tamper with the data or replaying old packets are detected using the Merkle-Trees, and thus will fail the integrity verification.

Finally, assigning global memory region to a processing element requires transferring the encryption key from the processing element managing the global memory to the requester. The key exchange process should be done in a secure key exchange mechanism as used in previous work [9] to exchange keys between the processor and the memory.

5. Methodology

To evaluate our scheme, we used the Structural Simulation Toolkit (SST) [23]. We implemented the BMT and the ToC, and the security metadata caches. We added latencies to model the overhead of encryption. We modified the memory controller to handle the encryption and integrity verification. The configuration of the modeled system are listed in Table 1.

To stress our proposed scheme, we used memory intensive applications from SPEC2006 [13] benchmarks, and some HPC workloads such as Lulesh2.0 [15], miniFE.x [14], pennant [10], and SimpleMOC [12]. We run each application for 500M instructions using a single thread for SPEC2006 applications and four threads for other workloads. We run the experiments using a single processing element as using multiple processing elements sharing the memory can result in security metadata coherence problem, which is beyond the scope of this work. We implemented a baseline scheme that uses a single Merkle-Tree protecting the global memory region and does not protect the local memory. We implemented a scheme that uses a single Merkle-Tree to protect both memories and does not partition the security metadata cache. Then, we compared

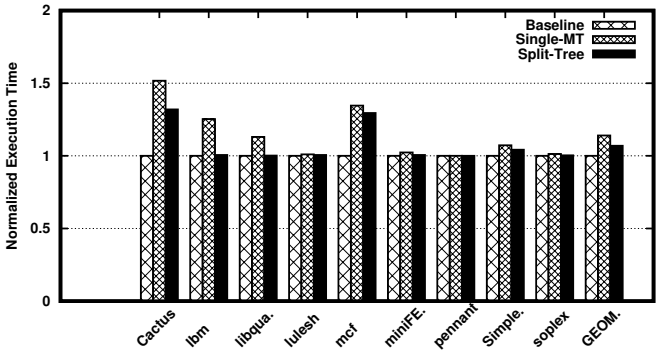

Figure 8. Split-Tree Impact on Performance.

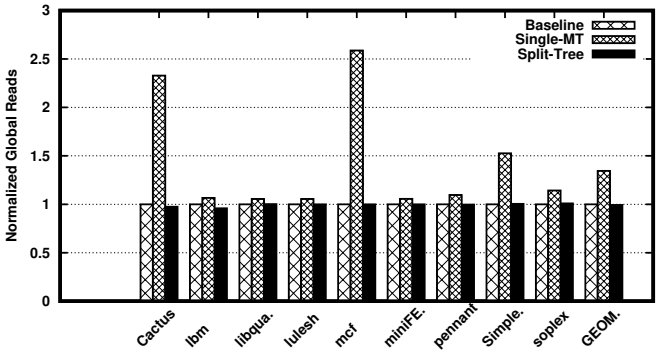

Figure 9. Split-Tree Impact on Global Memory Reads.

the single-MT scheme and our proposed split Merkle-Tree scheme with the baseline. Finally, we implemented the cache partitioning and used a DRAM cache region.

6. Evaluation

To evaluate our scheme, we implemented the Split-Tree scheme, a Single-MT scheme, DRAM caching for global MT nodes, and security metadata cache partitioning. We compared the results with a baseline of a system that only protects the global memory with one tree.

6.1. Split-Tree Impact on Performance

Figure 8 shows the performance overheads of Single-MT scheme, and Split-Tree scheme compared to the baseline. The Single-MT scheme has an average performance overhead of 14%, which spikes to reach 51% for *cactus*, 34% for *mcf*, and 25% for *lbm*. This spike can be explained by the increase in global memory requests observed by these applications in the Single-MT scheme as shown in Section6.3 and Section 6.2. On the other hand, the Split-Tree scheme has an average performance overhead of 6.9% which spikes to reach 31% for *cactus*, and 29.5% for *mcf*. The overheads are caused by the Split-Tree local memory accesses as discussed in Section 6.2 and Section 6.3. Note that the Split-Tree scheme is reducing the performance overhead by making the requests going to the local memory which has faster accesses than the global memory.

6.2. Split-Tree Impact on Memory Reads

As shown in figure 9, the Single-MT scheme has an average of 34.5% reads to the global memory. These reads are caused by the security metadata misses which are required to verify the integrity of the required data cachelines. On the other hand, the Split-Tree scheme has no extra reads to the global memory, in a matter of fact, Split-Tree scheme reduces the global memory reads by an average of 1%, which is caused by verifying the integrity of the required data cacheline using the local

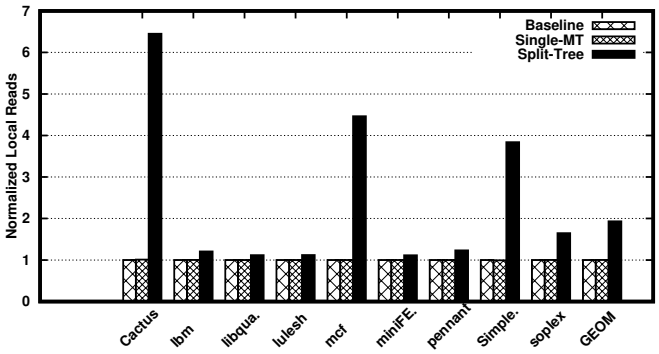

Figure 10. Split-Tree Impact on Local Memory Reads.

memory MT. However, the Split-MT scheme reduces the global memory reads by increasing the local ones.

As shown in figure 9, the Single-MT global memory reads spikes to reach 259% for *mcf*, and 232% for *cactus* which explains the high performance overhead for these applications. On the other hand, Split-MT scheme has 5% less global memory reads for *lbm* which explains the huge performance improvement for this application in the Split-MT scheme.

Figure 10, shows the normalized read operations by the schemes. We observe that Single-MT scheme has no effect on the local memory reads, which is expected as the Single-MT scheme protects the local memory using the same MT protecting the global memory, and as a result the security metadata misses are fetched from the global memory as discussed earlier.

The Split-MT scheme has 193% local memory reads on average, which spikes to reach 644% for *cactus*, 446% for *mcf*, and 383% for *SimpleMOC*. The high number of local memory reads for these applications is explained by the low security metadata cache hit rate for these applications as discussed in Section 4.3. However, we notice that *SimpleMOC* does not have a high performance overhead as other applications having similar memory accesses, which is explained by the low number of memory accesses for this application.

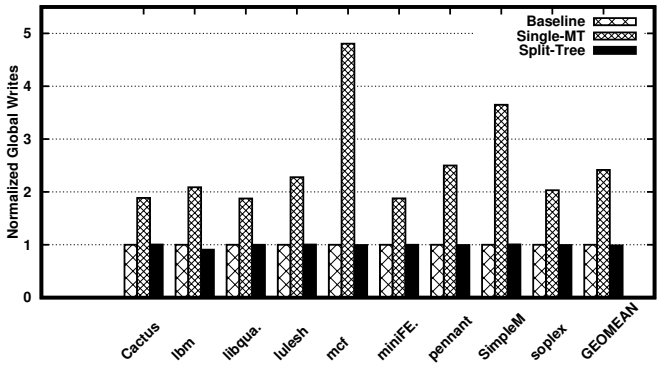

Figure 11. Split-Tree Impact on Global Memory Writes.

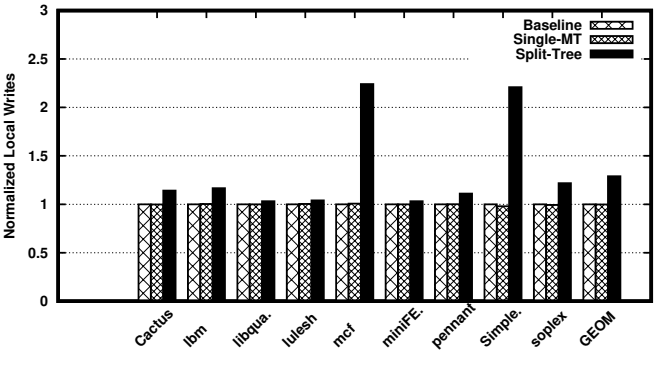

Figure 12. Split-Tree Impact on Local Memory Writes.

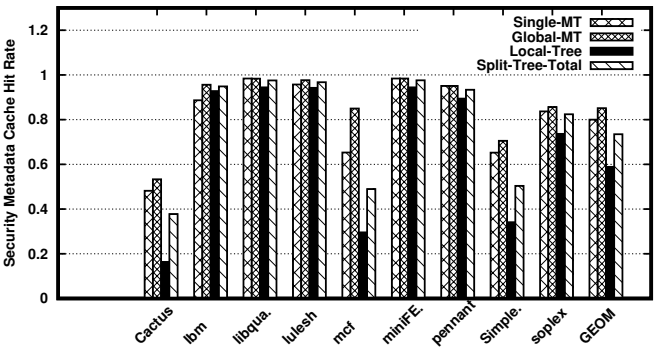

Figure 13. Security Metadata Cache Hit Rate.

6.3. Split-Tree Impact on Memory Writes

Figure 11 shows the global memory writes incurred by the schemes. The Single-MT scheme has an average of 241% writes, which spikes to reach 480% for *mcf*, and 365% for *SimpleMOC*. We observe that applications having the lowest security metadata cache hit rate are experiencing the highest extra memory accesses, and due to protecting the local memory using the same MT protecting the global memory, the accesses to fetch the required security metadata are directed to the global memory. On the other hand, the Split-Tree scheme has no extra writes to the global memory but reduces the writes to the global memory by 1.1% on average. We observe that *lbm* has 10% less writes to the global memory, which is caused by having a higher security metadata cache hit rate in the Split-MT scheme.

Figure 12 shows the local memory writes for the schemes. We observe that Single-MT scheme has no effect on the local memory writes as the security metadata writes are directed to the global memory. On the other hand, the Split-MT scheme has an average of 29% extra writes to the local memory which spikes to 224% for *mcf*, and 220% for *SimpleMOC* due to the low security metadata cache hit rate. We noticed that Split-MT scheme has two advantages over the Single-MT scheme in terms of writes. First, Split-MT changes the writes overhead to the local memory (DRAM) instead of the global memory (NVM), which translates into better performance and increases the lifetime of the NVM by 2.4x. Second, as the local memory has a smaller capacity and thus a smaller MT, the local MT updates are causing only 29% instead of the 124%.

6.4. Cache Partitioning Impact on Split-Tree

In order to analyze the effect of Split-Tree on the security metadata cache, we started by collecting the security metadata cache hit rate for the Single-MT scheme, the local MT in the Split-Tree scheme, and the global MT in the Split scheme, and finally the overall Split-Tree scheme.

Figure 13 shows the security metadata cache hit rate for the schemes. Single-MT scheme has an average security metadata cache hit rate of 80%, with *cactus* having a 48% hit rate, *mcf* and *SimpleMOC* having a 65.2% hit rate. On the other hand, the global MT in the Split-Tree scheme is showing a better security metadata cache hit rate for all the applications with an average of 85% hit rate. The hit rate of the MT is improved because the global MT is actually smaller than the Single-MT and it is only updated whenever data is written back to the NVM. However, the local MT is showing a very low average hit rate of 58.8% where *cactus* is having a 16.3%, *mcf* is having 29.5%, and *SimpleMOC* is having a 34.1% hit rates. Using a BMT for the local MT which is lazily updated makes the usage for the local MT nodes less frequent comparing to the eagerly updated global MT which results in evicting most of the local MT nodes. While caching the global MT nodes is more desirable but it is severely degrading the hit rate for the local MT. To analyze this low hit rate of the local MT, we collected the distribution of security metadata cache and analyzed the accesses to the MT nodes. We observed that without partitioning the security metadata cache, the global MT nodes are occupying 97.8% of the security metadata caching.

Figure 14 shows the access distribution for the MT levels. We observe that Single-MT and the global MT are showing a similar behavior where 25.5% and 29.5% of

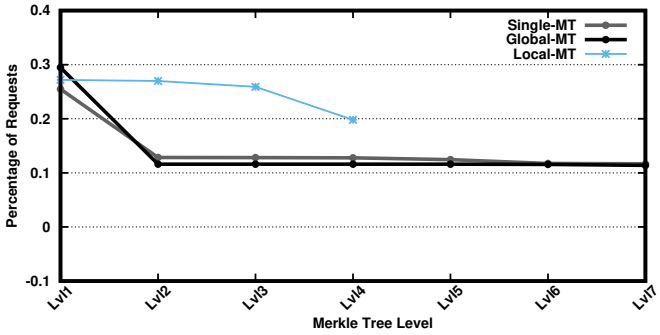

Figure 14. Access Distribution to Merkle-Tree Levels.

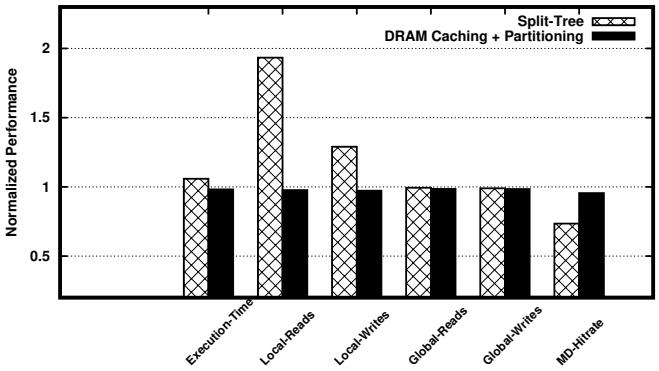

Figure 15. DRAM Caching and Partitioning Impact on Performance.

the accesses are going to the first level, then the number of accesses are saturating around 11%. The saturation level 11% represents the generated writes used to eagerly update the trees. The lower levels are showing higher accesses due to stopping the verification of read data at the first cache hit. On the other hand, the local MT is lazily updated and thus the higher MT nodes are only required when a dirty child node is evicted, or for read verification of the child node is missing. Due to this less frequent of the upper nodes they get evicted by the contesting eagerly updated global MT nodes. However, this is not the case with the Split-Tree scheme. As shown in Figure 14, the local MT has 4 levels which are receiving 27%, 26.9%, 25.9%, and 19.8%. Despite the use of the lazy update scheme, and the high node coverage of the BMT, the local MT lower levels nodes are getting evicted which requires requesting higher levels for read verification, which explains the access behavior of the local MT levels despite the use of the lazy update.

6.5. DRAM Metadata Caching and Cache Partitioning Impact on Split-Tree

To improve the system performance, we enabled DRAM caching of global MT nodes by allocating a 256KB region and using it as a cache. The DRAM caching improved the system performance slightly. Note that, a DRAM cached metadata miss will cause a higher latency to fetch the required global MT node, as the memory controller has to check the DRAM caching region first and then send the request to the global memory if the node is not cached in the DRAM cache region. To improve the performance further, we overlapped the requests by sending the request to the local memory as well as the global memory, and if the request was served from the DRAM, the global request is ignored. Finally, we applied cache partitioning techniques to prevent the local MT and the global MT from contesting over the cache resources. As the local MT is using a lazy update scheme and due to the small local memory size, we limited the local MT nodes partition to 30% of the security metadata cache size. Furthermore, we cached the lowest two levels of the local MT only and partitioned the global MT cache sets to prevent lower global MT levels from evicting higher MT levels, and assigned a smaller number of sets for higher levels. The MT node coverage increase as the level of the MT node increases, therefore we assigned one set for the highest two levels, and increased the number of sets for the lower levels dynamically.

Figure 15 shows all the performance aspects of the DRAM caching combined with cache partitioning compared to the Split-Tree. The cache partitioning and DRAM caching of security metadata improved the performance by 7%, local memory reads by 50%, local memory writes by 30%. We observe that, despite the use of DRAM caching which is expected to increase the number of local memory accesses, the number of accesses to the local memory decreased. This decrement is justified by the huge improvement in the security metadata cache hit rate, as the cache partitioning improved the hit rate from 73.5% to 95.5%.

7. Conclusion

Protecting the confidentiality and integrity of the data in FAM architecture is challenging and requires special handling due to having two different memories. Implementing secure memory architecture schemes directly can introduce higher overheads. Split-Tree is a scheme that uses a dedicated integrity tree to protect the local memory, and another integrity tree to protect the global memory. Using two different integrity trees can reduce the performance overhead of traditional secure memory implementation schemes. However, having two different trees will cause a high contest over the security metadata cache and can lead to unnecessary performance overheads and extra memory accesses. To reduce the effect of the contest over the cache resources, we partition the security metadata cache to have static partition for the local MT and another partition for the global MT. Furthermore, we prevent caching higher

levels of the local MT due to the use of lazy update for the local MT, and we partition the global MT cache sets between different MT levels dynamically.

Using Split-Tree can reduce the performance overhead by 7%, global memory reads by 34%, global writes by 140%. However, this improvement is achieved by increasing the local memory reads by 93%, and local memory writes by 29%. Finally, implementing cache partitioning techniques and allowing DRAM caching of the global MT nodes improved the performance by additional 7%, which is stemming from the high improvement of the security metadata cache hit rate.

Acknowledgement. This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Approved for public release. Distribution is unlimited.

References

[1] "CCIX specifications V 1.0," https://www.ccixconsortium.com/library/specification/, accessed: 2019-11-10.

[2] "CXL 1.1 specifications," https://www.computeexpresslink.org/, accessed: 2019-11-10.

[3] "GenZ specifications kernel description," https://genzconsortium.org/faqs/, accessed: 2019-09-30.

[4] "Intel® Optane™ DC Persistent Memory Operating Modes Explained," https://itpeernetwork.intel.com/intel-optane-dc-persistent-memory-operating-modes/#gs.xtmcnw, accessed: 2020-24-02.

[5] M. Alwadi, A. Mohaisen, and A. Awad, "Phoenix: Towards persistently secure, recoverable, and nvm friendly tree of counters," 2019.

[6] A. Awad, P. Manadhata, S. Haber, Y. Solihin, and W. Horne, "Silent shredder: Zero-cost shredding for secure non-volatile main memory controllers," *ACM SIGOPS Operating Systems Review*, vol. 50, no. 2, pp. 263–276, 2016.

[7] A. Awad, Y. Solihin, L. Njilla, M. Ye, and K. Zubair, "Triad-nvm: Persistency for integrity-protected and encrypted non-volatile memories," in *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 2019, pp. 169–180.

[8] 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, pp. 610–614.

[9] A. Awad, Y. Wang, D. Shands, and Y. Solihin, "Obfusmem: A low-overhead access obfuscation for trusted memories," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 107–119.

[10] C. R. Ferenbaugh, "Pennant: an unstructured mesh mini-app for advanced architecture research," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 4555–4572, 2015.

[11] S. Gueron, "A memory encryption engine suitable for general purpose processors," 2016, https://eprint.iacr.org/2016/204.

[12] G. Gunow, J. Tramm, B. Forget, K. Smith, and T. He, "Simplemoc-a performance abstraction for 3d moc," 2015.

[13] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, sep 2006. [Online]. Available: https://doi.org/10.1145/1186736.1186737

[14] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, vol. 3, 2009.

[15] I. Karlin, J. Keasler, and J. Neely, "Lulesh 2.0 updates and changes," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2013.

[16] V. R. Kommareddy, S. D. Hammond, C. Hughes, A. Samih, and A. Awad, "Page migration support for disaggregated non-volatile memories," in *Proceedings of the International Symposium on Memory Systems*, 2019, pp. 417–427.

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

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

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

[20] Z. Li, R. Zhou, and T. Li, "Exploring high-performance and energy proportional interface for phase change memory systems," *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 210–221, 2013.

[21] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch, "System-level implications of disaggregated memory," in *IEEE International Symposium on High-Performance Comp Architecture*. IEEE, 2012, pp. 1–12.

[22] S. Liu, A. Kolli, J. Ren, and S. Khan, "Crash consistency in encrypted non-volatile main memory systems," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 310–323.

[23] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls *et al.*, "The structural simulation toolkit," *SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 37–42, 2011.

[24] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "Using address independent seed encryption and

bonsai merkle trees to make secure processors os-and performance-friendly," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007, pp. 183–196.

[25] B. Rogers, M. Prvulovic, and Y. Solihin, "Efficient data protection for distributed shared memory multiprocessors," in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. ACM, 2006, pp. 84–94.

[26] B. Rogers, C. Yan, S. Chhabra, M. Prvulovic, and Y. Solihin, "Single-level integrity and confidentiality protection for distributed shared memory multiprocessors," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. IEEE, 2008, pp. 161–172.

[27] G. Saileshwar, P. Nair, P. Ramrakhyani, W. Elsasser, J. Joao, and M. Qureshi, "Morphable counters: Enabling compact integrity trees for low-overhead secure memories," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 416–427.

[28] 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)*. IEEE, 2018, pp. 454–465.

[29] 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*. ACM, 2018, pp. 665–678.

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

[31] M. Ye, K. Zubair, A. Mohaisen, and A. Awad, "Towards low-cost mechanisms to enable restoration of encrypted non-volatile memories," *IEEE Transactions on Dependable and Secure Computing*, 2019.

[32] S. F. Yitbarek, M. T. Aga, R. Das, and T. Austin, "Cold boot attacks are still hot: Security analysis of memory scramblers in modern processors," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 313–324.

[33] K. A. Zubair and A. Awad, "Anubis: ultra-low overhead and recovery time for secure non-volatile memories," in *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 2019, pp. 157–168.

[34] 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, pp. 479–492.