Compact Leakage-Free Support for Integrity and Reliability

Meysam Taassori
University of Utah
Salt Lake City, USA
taassori@cs.utah.edu

Rajeev Balasubramonian
University of Utah
Salt Lake City, USA
rajeev@cs.utah.edu

Siddhartha Chhabra
Intel
Hillsboro, USA
siddhartha.chhabra@intel.com

Alaa R. Alameldeen
Intel
Hillsboro, USA
alaa.r.alameldeen@intel.com

Manjula Peddireddy
Intel
Santa Clara, USA
manjula.peddireddy@intel.com

Rajat Agarwal
Intel
Hillsboro, USA
rajat.agarwal@intel.com

Ryan Stutsman
University of Utah
Salt Lake City, USA
stutsman@cs.utah.edu

***Abstract*—The memory system is vulnerable to a number of security breaches, e.g., an attacker can interfere with program execution by disrupting values stored in memory. Modern Intel® Software Guard Extension (SGX) systems already support integrity trees to detect such malicious behavior. However, in spite of recent innovations, the bandwidth overhead of integrity+replay protection is non-trivial; state-of-the-art solutions like Synergy introduce average slowdowns of 2.3× for memory-intensive benchmarks. Prior work also implements a tree that is shared by multiple applications, thus introducing a potential side channel. In this work, we build on the Synergy and SGX baselines, and introduce three new techniques. First, we isolate each application by implementing a separate integrity tree and metadata cache for each application; this improves metadata cache efficiency and improves performance by 39%, while eliminating the potential side channel. Second, we reduce the footprint of the metadata. Synergy uses a combination of integrity and error correction metadata to provide low-overhead support for both. We share error correction metadata across multiple blocks, thus lowering its footprint (by 16×) while preventing error correction only in rare corner cases. However, we discover that shared error correction metadata, even with caching, does not improve performance. Third, we observe that thanks to its lower footprint, the error correction metadata can be embedded into the integrity tree. This reduces the metadata blocks that must be accessed to support both integrity verification and chipkill reliability. The proposed Isolated Tree with Embedded Shared Parity (ITESP) yields an overall performance improvement of 64%, relative to baseline Synergy.**

***Index Terms*—Memory systems, security, integrity verification**

I. Introduction

The memory system is vulnerable to a wide range of attacks. One class of memory system attacks, referred to as replay attacks, tries to modify memory contents, thus disrupting the victim program's execution. Such attacks can be carried out by a compromised OS, by an attacker with physical access to the hardware, by co-scheduled threads, or by malicious agents in the supply-chain. It is clear that privileged execution (by a compromised OS) or a custom memory module can modify any memory location; even a user-level co-scheduled thread can modify the victim's memory space with a row hammer attack [36], [21]. More recently, it was alleged that malicious chips in the motherboard have been used to implement a man-in-the-middle attack that manipulates memory responses and disrupts how an OS boots up [31]. Given these many attack possibilities, it is important that a secure system verify the integrity of data being fetched from external sources. Integrity verification has therefore also been incorporated into the Memory Encryption Engine (MEE) used in implementations of Intel® Software Guard Extensions (Intel® SGX) [25], [9], [12].

Integrity verification incurs a significant performance penalty. For workloads with small working sets, integrity verification in MEE can impose a penalty of 1.8×, while larger workloads can suffer slowdowns of over 5× [40], [29], [3]. In typical implementations, a data block is verified by checking its associated message authentication code (MAC). To prevent the attacker from replaying an older message and an older MAC, the generation of the MAC involves a version number or counter. A tree of hash functions is then constructed over the counters [32], [12] and verified on every access. Integrity verification (including protection from replay attacks) therefore imposes the following overheads on every data block access: fetching the MAC, fetching the counter, and fetching the integrity tree ancestors of the counter. While some of these metadata structures can be effectively cached and the latency hidden with speculation [23], [22], [34], it is the memory bandwidth overheads of these additional accesses that contribute to most of the slowdown from integrity verification.

Recent state-of-the-art solutions include VAULT [40], Morphable Counters [33], and Synergy [34]. In particular, Synergy makes the observation that if the system uses ECC DIMMs, an integrated solution for reliability and integrity can offer lower overheads. It places the MAC in the space usually reserved for ECC. This allows the MAC to be fetched to the processor without requiring a separate memory transaction. The MAC is also effective at detecting run-time soft and hard errors with a very high probability. To correct any discovered errors, a separate parity field per data block is maintained. While this

parity represents a storage overhead similar to the MAC, it is only accessed when the corresponding data block is written, not when the data block is read. Synergy thus helped reduce the average slowdown from 2.55× in VAULT to 2.3×.

Even though Synergy provided a significant advancement for memory integrity solutions, it still suffers from the following issues: (i) a single integrity tree protects the entire physical memory, which leads to inter-application interference, (ii) every data block write requires a parity update in memory, and (iii) the parity update requires DRAM write masking which is not supported on all systems. We dig into each of these drawbacks next.

Our study first analyzes the nature of metadata overheads imposed by modern implementations of integrity verification. When a single integrity tree is constructed for all pages in physical memory, a node in the integrity tree has descendants that can belong to different applications. We show that this leads to reduced locality and higher interference in the metadata cache. We demonstrate in Section III-B that the shared resources also create potential side channels. We solve these issues by implementing a separate integrity tree and metadata cache for each enclave. This requires a new level of indirection, mapping enclave pages to consecutive leaves in its tree.

While the above approach yields a lower metadata cache miss rate, our analysis shows that metadata misses are typically correlated, i.e., we often incur a miss for both the leaf node and the MAC, resulting in two memory fetches. To combine these memory fetches into one, we exploit an opportunity offered by Synergy. We first reduce the parity overhead in Synergy by sharing the parity among multiple data blocks in different memory ranks. This preserves the chipkill protection of Synergy with very high probability, while reducing the parity footprint. However, updates to the parity field now require read-modify-writes, similar to RAID-5. Therefore, shared parity by itself is not able to improve upon the Synergy approach. We then observe that with shared parity, its footprint is small enough that it can be embedded in the integrity tree. A leaf node in the tree is modified so it handles half as many counters; this creates enough room to store the shared parity for the corresponding data blocks. Thus, a single leaf node fetch provides both counters *and* parity for a data block. A neat side effect of our approach is that unlike Synergy, we do not need DRAM write masking, which is not supported in all commercially available systems. Our solution thus addresses all three of the problems we identified in Synergy.

We carry out a detailed exploration of the design space for the proposed *Isolated Tree with Embedded Shared Parity (ITESP)*, considering different baselines, address mapping policies, integrity trees, etc. The primary contributions are:

- We show that an MEE-like shared integrity tree can lead to inefficiency and leakage in the metadata cache.
- With a new level of indirection, we introduce isolated trees and metadata caches per enclave to eliminate this side channel and improve metadata cache hit rate. Such isolation improves the performance of Synergy by 39%.
- We then augment Synergy with parity sharing and parity caching. While this significantly reduces metadata storage, it slightly degrades performance because of parity read-modify-write operations.
- We then design ITESP by including shared parity in the integrity tree. The unified data structure leads to a lower penalty for metadata cache misses and boosts the improvement over Synergy to 64%.
- We quantify the negligible impact of ITESP on reliability. We show that it offers the same integrity guarantees as the baseline. By avoiding write masking, ITESP is compatible with more systems. We confirm significant performance and energy improvements for a variety of system configurations for 31 benchmarks drawn from 3 suites.

II. Background

A. Threat Model

We assume a threat model and security guarantee similar to popular memory security techniques [25], [9], [12]. As in MEE, a region of memory or an "enclave" is assigned to an application with *confidentiality* and *integrity* guarantees. In this paper, a guarantee of integrity includes protection from replay attacks. The application's enclave is thus protected from integrity attacks from a compromised OS, from co-scheduled threads, and from modified hardware components. The memory controller provides confidentiality with encryption/decryption when accessing data in the enclave. Integrity support is more complicated.

Integrity guarantees that the data returned from memory matches the last write to that location. Integrity of data can be verified by confirming its MAC. If an attacker has the ability to precisely control a block, they can engage in a replay attack, where they feed the processor an earlier valid block/MAC combination. For software attacks that cause random bit flips, e.g., row hammer, a MAC per block is enough to provide integrity protection. Gueron [12] states that the MEE threat model includes physical attacks where a malicious memory module can perform a replay attack by precisely returning an older block/MAC; to defend against such hardware attacks, as well as similar software attacks, an integrity tree is required.

In addition to integrity support, we introduce defenses against a limited set of side channels. Co-scheduled applications on a processor share a number of resources; such sharing introduces side channels and vulnerabilities, e.g., in data/instruction caches [45], [20], branch predictors [10], coherence directories [48]. As we show later, the shared integrity tree and metadata cache can also be exploited by a co-scheduled attacker to establish a side channel and leak sensitive information from a victim program; we defend against this particular side channel. The many other side channels in the system will have to be defended by other complementary techniques [30], [46], [48].

B. Integrity Verification

To support integrity, every data block is associated with a MAC, which is essentially a *keyed hash* of the data block. If an attacker tampers with data, the hash on the new block will likely not match the MAC retrieved from memory. To prevent replay attacks, where an attacker returns an old version of data/MAC, every data block is associated with a version number or counter that is used in the encryption function. An integrity tree is formed where the counters form the leaves and every parent node is a hash of the child nodes. To confirm that a correct counter has been returned from memory, the ancestor nodes of the counter are fetched until a cache hit is encountered, and the hashes are confirmed.

We assume a data block size of 64 bytes, a MAC per data block of 8 bytes, and an ECC per data block of 8 bytes. On an ECC DIMM, a 64B data fetch is accompanied by the 8B ECC; the MAC is fetched with a separate memory transaction that brings in 8 MACs for 8 consecutive data blocks (and its ECC). Similarly, each node of the integrity tree is 64B (plus ECC) and requires a separate memory transaction.

SGX uses a different integrity tree organization, called MEE [12], where the linkage between parent and child node is formed by hashing the child node and a counter in the parent node; the hash is then placed in the child node. This approach offers higher arity and therefore a more compact tree. VAULT [40] improves upon the MEE integrity tree organization by decomposing the counter in a node into a small local counter and a larger shared counter (an idea also used in the BMT [32]). This further improves the arity of the tree and shrinks its depth. Morphable Counters [33] observes locality in counter values and adjusts the shared global counter value; this keeps the overflow rate low even for few-bit local counters. The smaller local counter size leads to higher arity. In our analysis, we assume baselines with integrity trees modeled after both VAULT and Morphable Counters.

Most prior work has cached parts of these additional data structures in the LLC or separate caches [22]. The MACs exhibit limited temporal locality, i.e., if the data block has a miss in the LLC, its MAC is also likely to miss in a MAC cache. But the MAC cache can exploit spatial locality because a single 64-byte entry in the MAC cache accommodates MACs for eight consecutive cache lines. Similarly, an integrity tree cache entry also exploits spatial locality. Misses for lower levels of the tree (leaf nodes) are much more likely than misses for higher levels.

The key takeaway is that in addition to the data blocks themselves, two additional data structures have to be managed: the MAC per block and the integrity tree. When the data block is not found in the LLC, barring spatial locality opportunities, there is a high chance that the block's counter and MAC will also not be found on-chip, thus requiring two more memory fetches.

C. Synergy

In the Synergy proposal, Saileshwar et al. [34] observed that enterprise systems providing integrity guarantees are also likely to provide high reliability with ECC DIMMs. Thanks to the additional storage and bandwidth in such DIMMs, every 64-byte data block transfer is accompanied by 8 bytes of metadata. Instead of placing ECC in the 8-byte metadata field, Synergy places the MAC in that field. Figure 1 shows how data and metadata are organized for a baseline memory system and for Synergy [34]. In the baseline, the 8-byte metadata field accompanying every 64-byte data block is responsible for error detection and correction, while the data block's 8-byte MAC is stored elsewhere in memory.

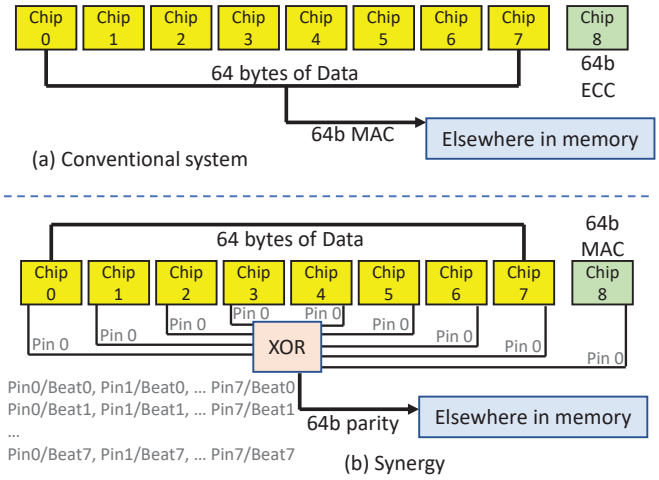


Fig. 1: *Data organization in baseline memory and Synergy.*

In Synergy, the 8-byte field accompanying every 64-byte data block stores the MAC. When a block is read, integrity verification is enough to confirm with a very high probability that the block is free of soft and hard errors. When an error does occur, separate metadata stored elsewhere in memory is required for correction. Synergy implements this as a 64-bit parity field, as shown in Figure 1. The first parity bit captures the parity of pin 0 from all DRAM chips in the rank for the first beat[1]; the other 63 parity bits similarly capture other pins and/or other beats. This enables recovery for up to 8 pins, i.e., recovery is possible for an entire $\times 8$ chip[2] failure (chipkill). The correction procedure walks through every failure possibility until the corrected block has a matching MAC. While the latency of each correction is high, its overall impact on performance is negligible given the very low DRAM error rates [38], [39].

When writing a block, its parity also has to be updated. This requires a separate write to another memory location. DDR protocols allow write-masking, i.e., it is possible to only modify 64-bits of a 64-byte line, but such a transaction requires that the memory channel be occupied for all 8 beats, as if a 64-byte line is being written. A write in Synergy is therefore no more efficient than the baseline which also requires an

[1]A rank is the set of DRAM chips involved in accessing one data block. DDR memory systems transfer data at both rising and falling edges of the clock. Each edge is referred to as a beat.

[2]A $\times 8$ chip has 8 in/out data pins and handles 8 bits on every clock beat.

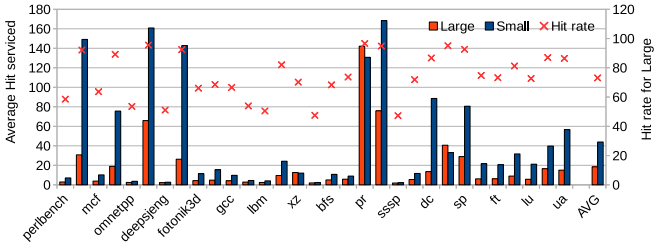


Fig. 2: *Metadata block utilization while in cache in VAULT (left Y-axis) and metadata cache hit rate (right Y-axis).*

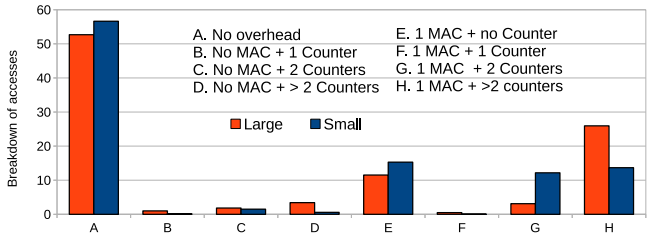


Fig. 3: *Breakdown of metadata access patterns.*

additional 64-bit write for the MAC. Synergy's improvement stems from its efficient reads; the MAC is included in the 72 bytes fetched on a block access, and it is used for both integrity verification and error detection.

The Synergy approach relies on write masking so that only the parity bits of a modified block can be updated. Some systems [47], [15] disable write masking for ECC DIMMs, presumably to allow a class of ECC where the data/ECC code span multiple beats. Write masking is also disabled in DIMMs that employ $\times 4$ chips because of restrictions on how DIMM pins are shared [26]. Thus, while Synergy works correctly and effectively in theory, it is not compatible with all systems. Our proposed solution does not require write masking.

D. Motivation

A key drawback in prior work is that they implement a single integrity tree for all physical pages, leading to inter-application interference. They also implement separate data structures for the tree and for MAC/parity, leading to more memory accesses. We analyze these two drawbacks here.

We first assume a VAULT baseline because it stores both MAC and tree in the metadata caches. Details of the simulation methodology are in Section IV. We consider two models here: (i) Large, where the integrity tree is constructed on the entire 128 GB physical memory and 4 programs are executed with a shared 64 KB metadata cache, and (ii) Small, where we assume a single program mapped to 32 GB physical memory and 16 KB metadata cache.

Figure 2 quantifies how metadata block utilization goes up significantly as we move from the Large to the Small model. The left Y-axis shows the hits per metadata block for the two configurations, while the right Y-axis shows the metadata cache hit rate for the Large model. On average, we see that the usefulness of a metadata block is 2.1$\times$ lower in the Large multi-programmed model. This is because of two reasons: less spatial locality per block because it often includes metadata from multiple programs, and conflict misses caused by multi-programmed interference.

Figure 3 shows the nature of metadata accesses triggered for every data block miss in the LLC. For both models, a significant fraction of data misses do not trigger any metadata accesses because of spatial locality. For another significant fraction (about 30% for both Large and Small) of data misses, both MAC and counter are not found in the metadata cache, i.e., these misses are usually correlated. The distribution of these correlated misses is a little different in both models. In Large, the higher levels of the tree experience more misses because of multi-programmed interference (shown in Case H), whereas in Small, the leaf node miss is usually accompanied by a miss for the parent alone (Case G), or parent and grandparent (Case H) nodes.

We observed similar trends in Synergy as well (not shown for space reasons). This analysis helps us understand the two main causes of metadata overheads. The take-home from Figure 2 is that inter-program interference greatly diminishes the utility of metadata cache entries. Figure 3 shows that because of multiple separate metadata structures, data block misses often require multiple metadata memory accesses.

III. Isolated Tree with Embedded Shared Parity

A. Isolated Metadata

Problem. MEE implements a single integrity tree for its Enclave Page Cache (EPC) region of physical memory, including pages from multiple enclaves. This can lead to inefficiency in the metadata cache because of inter-program conflict misses and because of reduced spatial locality per node. A second problem is that shared metadata and resources can be used to establish a potential side channel between two programs.

Consider the following simple covert channel established between two programs A and B. Program A can touch a set of pages, thus bringing their counters into the metadata cache. When Program B runs, it touches a set of pages such that their counters displace some of A's counters from the metadata cache[3]. When A runs, it touches the same set of pages again and uses the access latency to determine the displaced set of counters. Depending on the fidelity of the measurements, every such exchange can transmit a few bits of information between the two programs. In addition to using the shared metadata cache, the shared tree can also be used to exchange information. When a local counter in a tree node overflows, the global counter is updated, and all child nodes have to be read and re-encrypted. If A and B share counters in a node, A can issue a series of writes to its block, triggering a local counter overflow and a re-encryption process for all blocks handled by that node, including those belonging to B. B detects this action of A when it tries to access its block and experiences a longer

[3]Similar to prior attacks [14], this assumes that in a prior setup process, the two programs have synchronized their timing and identified pages that create conflicts with each other in the metadata cache.

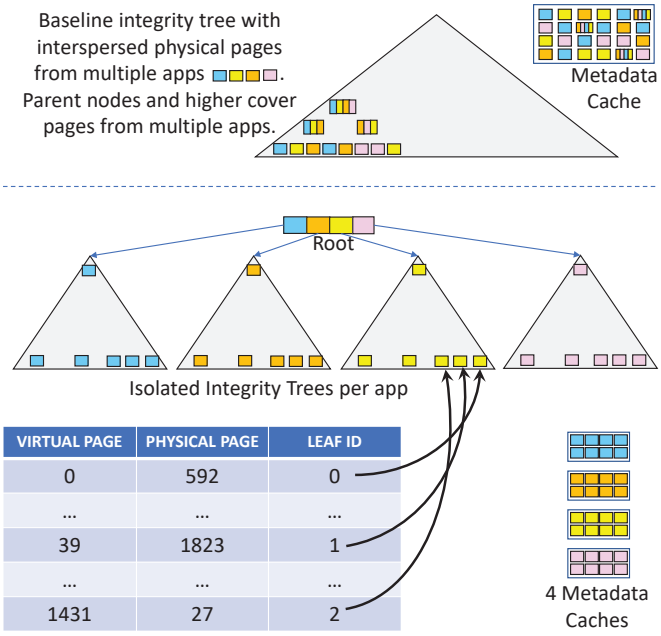


Fig. 4: *Baseline integrity tree and metadata cache for 4 apps (top). Isolated integrity trees and metadata caches (bottom).*

than usual latency because it has to wait for re-encryption to complete. Thus, a shared tree and shared metadata cache can both create separate side channels.

Proposed Solution. To address the efficiency and leakage problems, we propose to implement isolated integrity trees and metadata caches for each protected enclave in the system. Figure 4 shows the tree and metadata cache for the baseline and proposed approaches. The isolated trees only share the root node that always remains on the processor. In the baseline system, the physical page number is used to determine the integrity tree leaf-id for that page. Since physical pages of different enclaves are inter-mingled, we can no longer use the physical page number to determine the page's leaf-id. We can also not use the virtual page number because an enclave may be composed of multiple threads with private and shared pages, both of which can cause different forms of aliasing. We must therefore explicitly assign leaf-ids to every physical page within an enclave. This leaf-id assignment process is rolled into the duties of the memory management unit; when it assigns a free physical page to an enclave, it also assigns a free leaf-id to that page, which is then tracked in the page table and TLB. Since leaf-ids are assigned in the order that pages are touched, they benefit from the locality exhibited in their virtual addresses. When pages are reclaimed, the list of free leaf-ids is also updated. Further, the metadata cache is partitioned into private metadata caches, one per enclave. The enclave-id is used to access a specific partition of the metadata cache.

Depending on the integrity tree implementation and page interleaving across multiple memory channels, the counters for the blocks in a physical page may be mapped to multiple leaf nodes in a tree. The leaf-id is a pointer to the first such node and the page offset is used to compute the exact location of a counter for each block in that page.

With this first extension to the baseline, we improve locality in the metadata cache, while also eliminating two sources of side channels.

B. Covert Channel Demonstration

In this section, we demonstrate how a shared integrity tree can be exploited to establish a covert channel between two colluding processes. As we describe shortly, this approach can form the basis for a number of attacks. Our experimental platform uses an SGX v1 Intel®i5-7500 CPU at 3.4 GHz and Linux kernel version 4.15.0-54-generic[4]. Other cache- and memory-based side-channels are easier to exploit on this system. This new attack may only become relevant if other higher-bandwidth side-channels are addressed. Thus, this work adds to the growing list of known side channels that future secure processors must attempt to eliminate.

In our setup, the pages of an attacker enclave and victim enclave are interleaved, i.e., integrity tree nodes (Level 1 and above [12]) are shared by both enclaves. The attacker enclave first fills the metadata cache with irrelevant entries, the victim enclave then executes, followed by the attacker enclave. If the attacker enclave experiences a low latency, it implies that several metadata cache hits were encountered, i.e., the victim enclave touched a number of pages and warmed up the metadata cache with entries shared by both enclaves. This establishes a channel between the victim and attacker: a "1" is transmitted when the victim is memory-intensive and the attacker experiences low latencies, while a "0" is transmitted when the victim is non-memory-intensive and the attacker experiences high latencies. This is demonstrated in Figure 5 and we describe the methodology details below.

To enable different page placements within the EPC, we modify the kernel module to initialize the free list in a specific order. Upon enclave creation, the requested pages are then mapped to the specific intended locations within the EPC. We developed a high resolution timer for SGX to take measurements. As shown in Figure 5A, the attacker places a dummy data structure D at one end of the EPC. Accesses to D are used to clear other relevant entries from the metadata cache. The attacker has another data structure A, while the victim has a data structure V. The physical pages of A and V are interleaved.

Figure 5A shows the latencies experienced by the attacker when the victim is transmitting 0 (blue line) and 1 (red line). We take 10 measurements and show the range of measured latencies. The graph also varies the number of blocks touched by the victim and attacker on the X axis. By touching more blocks per measurement, we improve the fidelity of the data transmission, i.e., the latency ranges are less noisy and do not overlap, but this also reduces the covert channel bandwidth.

[4]Performance results are based on testing as of 2/14/2020 and may not reflect all publicly available security updates. No product can be absolutely secure. For more complete information about performance and benchmark results, visit www.intel.com/benchmarks.

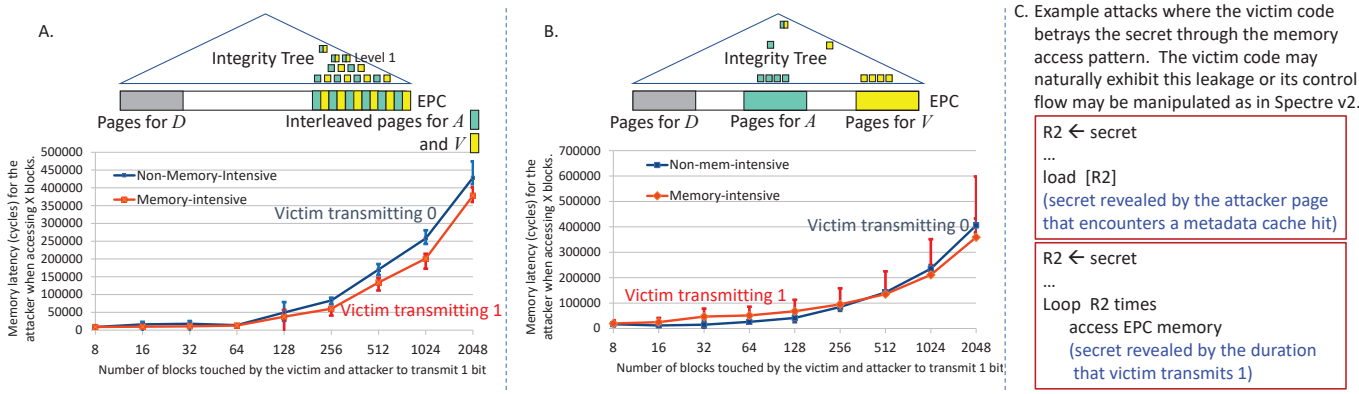


Fig. 5: *Covert channel demonstrated on an SGX v1 system with interleaved (A) or isolated (B) pages for the attacker and victim enclaves. Two example victim code vulnerabilities (C) are also shown.*

We see that by accessing 256 blocks, a reliable channel with 18 Kbps bandwidth can be established. This channel demonstrates that the performance of one enclave is affected by the behavior of the other enclave when they share the integrity tree and metadata caches.

Figure 5B shows how the red and blue lines effectively converge when the attacker and victim pages are not interleaved. Of course, this is not perfect isolation because the two enclaves will share higher level nodes of the tree. Since the OS is untrusted, we also cannot rely on the OS to isolate each enclave's physical pages. The previous section therefore introduces hardware-managed leaf-ids to implement isolated trees for each enclave.

The covert channel setup is a demonstration of a leakage mechanism. To expose secrets, a complete attack must be developed upon the leakage mechanism, and this attack can take many forms. For example, malware within a VM can use the covert channel to exfiltrate secrets to a co-scheduled VM. Alternatively, a victim program's memory intensity or memory access pattern can betray the secret (see examples in Figure 5C). Similar to the Spectre attacks [20], the attacker can force such leakage by causing the victim program to speculatively jump to leaky code after the secret is loaded in a specific register. In Spectre, the secret value is used as the address for a load; the resulting data cache miss within the attacker reveals the cache index and therefore the secret value. The first example in Figure 5C uses a similar approach, but the attacker detects a metadata cache hit on a specific page to learn the secret. The second example in Figure 5C executes a memory-intensive loop for a duration that is a function of the secret. The attacker measures the duration that it receives a 1 on its channel to learn the secret.

C. Caching Shared Parity

Problem. In Synergy, the parity field is updated on every write. This separate data structure incurs a significant overhead in terms of both memory capacity and memory bandwidth. It also requires Write Masking.

Parity Cache. A first approach to reducing the write overhead is to cache the parity fields in an on-chip parity cache. In case of spatial locality, an entry of the parity cache stores updated parity fields for consecutive blocks. When a block of parity fields is evicted from the parity cache, a single block write to memory can update the parity fields for up to 8 data blocks. Note that the parity cache is never updated by reads from memory, i.e., it simply serves as a coalescing write buffer. As we show later in Section V, adding such a 16 KB parity cache to Synergy yields an improvement of 3%.

Parity Sharing. In baseline Synergy, there is one 64-bit parity field for every 64-byte data block. To increase the effectiveness of the parity cache, we next try to increase the coverage of each parity field. We XOR the parity fields for N different blocks to yield one 64-bit parity field for $64N$ bytes of data. This lowers the storage overhead for parity and has the potential to improve its cacheability and degree of coalescing. The N data blocks sharing a parity field must be from different memory ranks, i.e., they do not share the same DRAM chip pins. With such an approach, when a block is read from a rank and an error is detected, we can correct the error while assuming that the other $N-1$ blocks sharing the parity are error-free. The error correction fails only when two+ independent errors happen simultaneously in similar locations of different ranks, which is a rare event. As we show with a detailed reliability analysis in Section III-G, even this impact can be mitigated.

Parity Read-Modify-Write. Shared parity has one significant drawback. Without sharing, the parity field for a data block being written can be directly computed. With sharing, the new parity field requires a read-modify-write, similar to how updates are done in RAID-3/4/5, i.e., the new parity field depends on the old parity field and the old/new values for the data block. Therefore, the parity cache must keep track of how a block's parity has changed; when the parity cache entry is evicted, it must read the old parity from memory, apply the parity diff, and then write the new parity back to memory. As we show in our results, even the high coverage of a shared parity cache cannot overcome the penalty of the parity read-modify-write.

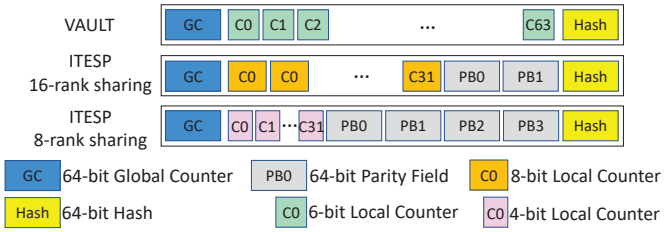


Fig. 6: *A block of counters in VAULT and* ITESP.

D. Embedding Parity in the Integrity Tree

Shared parity requires a read-modify-write and does not reduce bandwidth overheads. However, shared parity and its lower footprint have another benefit that we next exploit.

Opportunity. With sharing, the storage overhead for parity very closely resembles that for the counters. This presents an opportunity to create an effective combined data structure. Note that in all past systems, because of their different sizes, the two data structures (MAC/parity and counter/tree) are distinct, and separate memory fetches are required for the MAC/parity and for the counters/tree.

Counter+Parity. A block of counters in the VAULT baseline, shown in Figure 6, has a 64-bit shared global counter, 64 6-bit local counters, and a 64-bit hash. If we reduce the number of local counters, we can create room for a few parity fields. Figure 6 shows an example organization that has a 64-bit shared global counter, 32 8-bit local counters, a 64-bit hash, and 2 64-bit parity fields. If each parity field is shared by 16 data blocks, this metadata block contains both counter and parity information for 32 data blocks. Another relevant organization, also shown in Figure 6, is one with 32 4-bit local counters and 4 64-bit parity fields, while sharing parity among 8 blocks.

Such unification of counter and MAC/parity was not possible in prior systems where the MAC/parity had a larger overhead than the counters. While parity sharing by itself was not effective, it enabled a lower parity footprint and the effective unification of counter and parity metadata, which is a significant improvement over the Synergy baseline.

Larger Tree. The proposed change only applies to the leaf level of the tree. One downside is that the total number of leaf nodes in the tree has doubled. In essence, the tree has a larger footprint because it also packs in the parity information. We refer to this larger integrity/parity tree as ITESP. While this may impact the tree's cacheability, note that we now only need a single larger metadata cache per enclave instead of separate caches per enclave for counters and parity.

Higher Arity Baselines. The proposed organization also applies to other baselines, e.g., Morphable Counters. Figure 7 summarizes a Synergy-like baseline with Morphable Counters (SYN128) and two ITESP designs (ITESP64 and ITESP128), that introduce a trade-off between local counter overflow and metadata cacheability, which we quantify in Section V.

E. Implementation Details

Write Masking. Many systems [26], [47], [15] disable write masking for various DIMMs, i.e., Synergy cannot be deployed on all systems. ITESP performs counter/parity reads and writes at block granularity and does not require write masking.

Controller Complexity. ITESP implements a separate metadata cache per enclave. Once shared parity is embedded in the block of counters, it is fetched into the metadata cache, updated, and directly written into memory upon eviction. When a clean data block (as ascertained by the tag access) is being written, the block must be first read and "subtracted" out of its parity with an XOR operation. When a dirty data block is being evicted, it undergoes another XOR operation so it is "added" to the parity. In short, the parity update requires XORs involving the data block when it is first modified and when it is evicted. Isolated trees introduce an additional field in the page tables and TLBs to track the leaf-id for a physical page. The OS software that manages the list of free pages must also manage the list of free leaf-ids within each tree. A malicious OS cannot introduce a side channel through the integrity tree because the hardware uses the enclave-id to isolate each integrity tree.

Storage Overhead. ITESP reduces the storage requirements for metadata. Table I summarizes the metadata requirements for Synergy and ITESP. Parity sharing with ITESP is more helpful for DIMMs with $\times 16$ chips that need a larger parity for chipkill protection.

Organization	Integrity Tree	MAC/Parity	Total
VAULT	1.6%	12.5%	14.1%
Synergy128, $\times 8$ chips	0.8%	12.5%	13.3%
Synergy128, $\times 16$ chips	0.8%	25%	25.8%
ITESP64	1.6%	0	1.6%
ITESP128	0.8%	0	0.8%

TABLE I: *Metadata memory capacity overheads.*

Address Mapping. With ITESP, address mapping policies can noticeably impact performance. This is because the address mapping policy impacts row buffer locality, parallelism, and metadata cache locality. To exploit metadata cache locality, consecutive cache lines must share a global counter and parity. These consecutive cache lines must also be mapped to different ranks to enable chipkill. This means that blocks sharing a row buffer have a stride of N and yield a lower row buffer hit rate than the baseline. We evaluate these trade-offs in Section V and find an address mapping policy that balances these competing forces. In essence, four consecutive cache blocks are placed in a single bank and row buffer to promote row buffer hit rates. These four cache blocks must map to different parities. But since a leaf node may contain four parities, these blocks can share a leaf node, thus achieving a high metadata cache hit rate as well.

Metadata Cache Partitions. In this study, we assume that the metadata cache is uniformly partitioned across a fixed number of enclaves. To support a dynamic number of enclaves and varying working sets, additional hardware support similar to commercially available Cache Allocation Technology [28]

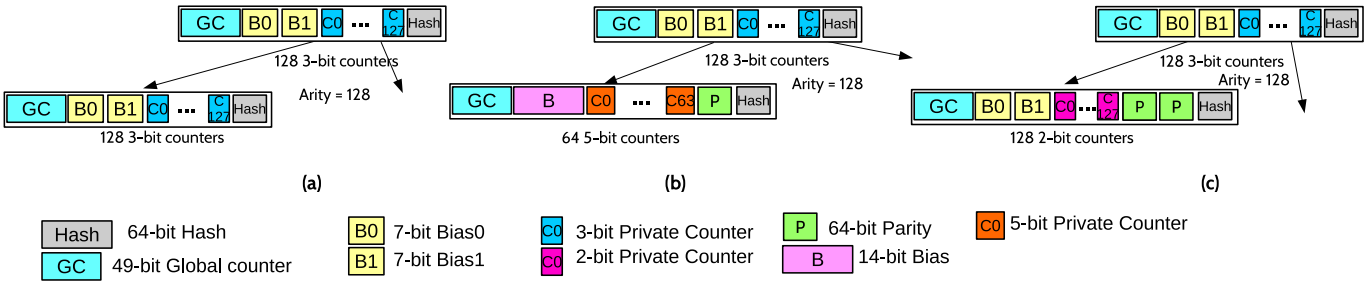


Fig. 7: *Different integrity trees with Morphable Counters: (a) SYN128: arity of 128 throughout, (b)* ITESP *64: arity of 64 at leaf level and 128 at other levels, (c)* ITESP *128: arity of 128 throughout.*

is required. The hardware, in addition to tracking various metadata per enclave, must also have registers to track the enclave's allocated range of metadata cache indices.

F. Security Analysis

The proposed changes have no impact on the system's integrity guarantees. Shared parity only has an impact on error correction capabilities, which is discussed in the next subsection. When parity is embedded in the tree, as shown in Figures 6 and 7, we are only changing the number of counters per node (and in turn, the number of nodes) or the size of local counters. Both of these changes impact efficiency, i.e., the metadata cache hit rate or the overflow rate, respectively. The size of the hash is unchanged, while the size of the overall counter remains in the 66-72 bit range, which negligibly impacts the hash collision rate.

We will state this more formally here, assuming that counters with 66-72 bits are equally effective at confirming hashes. Integrity is verified by confirming the following two equations: $MAC = f(Data, Counter, Key)$ and $Hash = g(Leafnode, Parentcounter, Key')$. Relative to prior work, ITESP does not make any change to the $Data$, $Counter$, Key, $Parentcounter$, and Key' fields. It modifies the organization of the $Leafnode$ by removing neighbor $Counter$ elements and adding $Parity$ fields. Because the $Leafnode$ includes the block's $Counter$ value (as in the baseline), the probability of identifying a replay attack is nearly identical to that of the baseline, i.e., any unexpected/malicious change to the $Counter$ will result in a highly likely $Hash$ mismatch. The $Parity$ field within the $Leafnode$ plays no role in detecting integrity violations; it can be viewed as padding before the $Leafnode$ is sent through the hash function.

As described earlier, when an integrity tree includes interspersed pages from multiple enclaves, a tree node has counters influenced by multiple enclaves. When one enclave accesses its data, a tree node (shared by multiple enclaves) may be brought to the metadata cache or may have to handle a local counter overflow. This affects the latency for another enclave that is also handled by the same node. The latency for an enclave's memory access is a function of (i) hits in its metadata cache, (ii) counter values in its own integrity tree, and (iii) contention at the memory controller. Isolation of the metadata cache and tree removes any leakage between enclaves from the first two sources. A shared memory controller is a separate potential leakage source (with or without integrity support) and will have to be eliminated with complementary techniques [44], [37], [11], [43], [42].

G. Reliability Analysis

Next, we evaluate the impact of shared parity on system reliability. When multiple blocks share parity, if the blocks are from the same rank, the parity function will involve multiple bits from each DRAM chip, thus preventing re-construction when a chip fails. Therefore, the multiple blocks must be from different ranks. If we assume that only a single chip can have an error (either hard or soft) at a time, the supported reliability is exactly the same as the baseline Synergy. When errors happen concurrently and independently on at least two different chips in a single rank, Synergy is unable to correct that error. The proposed ITESP approach also fails when errors happen concurrently and independently on at least two different chips in the entire memory. We are thus offering weaker reliability than Synergy in the case where at least two different chips in different ranks have concurrent and independent errors. Note that the probability of multiple independent errors is relatively small; further, if there is a background scrubbing process [35] that detects and corrects errors every few minutes, the probability of independent errors happening within a few minutes is even smaller. This is what we primarily quantify in Table II.

Error detection in ITESP and in baseline Synergy rely on the same MAC; so error detection capabilities are unaffected. Since multiple different blocks can hash to the same MAC (a *conflict*), there is a small probability of an error going undetected, leading to silent data corruption (SDC). Since the SDC rate of ITESP is the same as that of Synergy, we refer readers to the SDC analysis in the Synergy paper. To eliminate SDC for the common 1-bit error case, we can employ a 63-bit MAC and a 1-bit parity.

The analysis below assumes a scrub rate of 1 hour, i.e., concurrent independent errors are possible only when they manifest within the same hour. We base our DRAM failure rates on the empirical study of Sridharan and Liberty [38].

Case 1: SDC: A corrupted block with matching MAC during detection. The probability of a MAC conflict is 2^{-64}, i.e., less than 10^{-19}. A DRAM device has a FIT rate of

66.1 [38]. Assuming 288 DRAM devices in a memory system, this leads to an SDC rate of $288 \times 66.1 \times 2^{-64}$, i.e., less than 10^{-15} in every billion hours of operation. This rate is the same in both Synergy and ITESP.

Case 2: SDC: A corrupted block with matching MAC during correction. This happens when independent errors happen in two devices, the error is detected, and correction is declared a success because a matching MAC is found. For Synergy, the rate for a concurrent multi-device error in a rank is $288 \times 8 \times 66.1^2 \times 10^{-9}$ per billion hours, assuming a 9-device rank. The probability of a MAC conflict is 9×2^{-64} (since 9 MACs are computed during correction). The SDC rate would therefore be less than 10^{-20} in every billion hours of operation in Synergy. With ITESP, the probability of a multi-device error scales up linearly with the number of devices involved in correction. Assuming 288 devices in the memory system, the SDC rate would be less than 10^{-18} in every billion hours of operation.

Case 3: DUE: Multiple valid MACs during single error correction. When a single device fails, correction should be possible, but if more than one of the nine MAC checks succeeds, it is not possible to isolate the erroneous device and the error goes uncorrected. The rate for this occurrence is $288 \times 66 \times 8 \times 2^{-64}$, i.e., less than 10^{-14} per billion hours of operation. This is the same for Synergy and ITESP.

Case 4: DUE: Multi-chip error and no matching MACs. This is the common case during a multi-chip error, where all 9 MAC check attempts fail. The occurrence rate is that of two independent concurrent errors in the same rank in Synergy, i.e., $288 \times 66 \times 66 \times 10^{-9} \times 8$, less than 10^{-2} per billion hours. In ITESP, the two independent errors can happen in any two chips, for an occurrence rate of $288 \times 66 \times 66 \times 10^{-9} \times 287$, less than 1 per billion hours of operation.

Case	Synergy	ITESP
Case 1: SDC Rate	10^{-15}	10^{-15}
Case 2: SDC Rate	10^{-20}	10^{-18}
Case 3: DUE Rate	10^{-14}	10^{-14}
Case 4: DUE Rate	10^{-2}	1

TABLE II: *Summary of SDC and DUE rates per billion hours for Synergy and* ITESP.

Thus, Case 4 is the only noticeable degradation in reliability. While a single DUE per billion hours of operation for a memory system is already very low, it would be helpful to add more features that can reduce the error rate by two orders of magnitude, in line with that offered by baseline Synergy. One way to achieve that lower error rate is to trigger a scrub operation as soon as any error is detected (and likely corrected). Since every rank is typically accessed within a micro-second window, a chip-level failure will be detected within that window, triggering a correction and subsequent scrub. This would shrink the window for multi-error incidence from an hour to a few seconds, thus lowering its probability by three orders of magnitude.

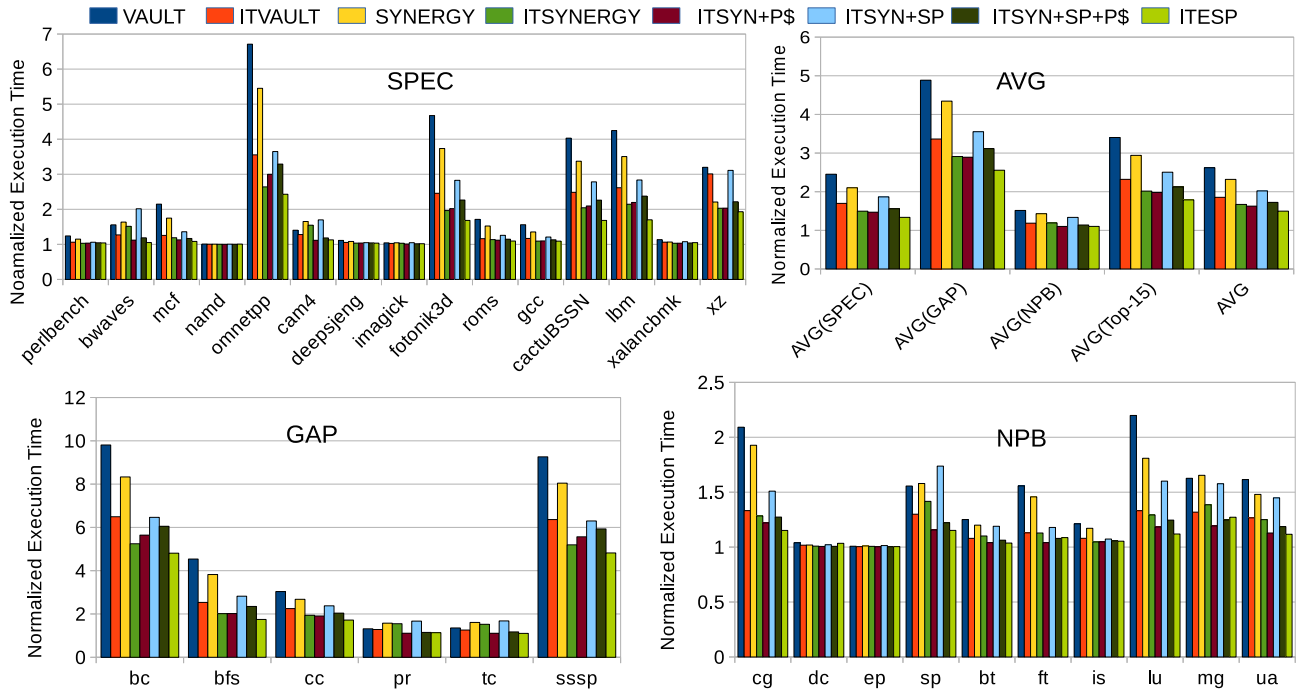
IV. Methodology

We evaluate our techniques on 31 workloads, including 15 from SPEC2017 [7], 6 from GAP [6], and 10 from NAS [5]. We use Pin [24] to generate virtual address traces for these workloads; we use page table dumps to convert these virtual address traces into physical address traces so we accurately capture how multi-programmed workloads have interspersed physical pages in the baseline. For most of our analysis, we focus on 4-program executions, where we execute 4 instances of the same program. After fast-forwarding to the region of interest and warming up the caches, we collect traces for five million memory reads and writes per program. The generated traces are fed to USIMM [8], a trace-based cycle-accurate memory simulator. Tables III and IV show the specifications of our simulator and our benchmarks, respectively. We also identify the 15 most memory-intensive benchmarks in Table IV that are the target of our proposed techniques. For a 128-arity tree, a local counter overflow incurs an overhead of 4K cycles. For our energy evaluation, we use Micron power calculator [1] to estimate the power of each memory chip. System energy is estimated with similar assumptions as the Memory Scheduling Championship that factor in CPU/memory utilization. Most of our results are for a 4-core system and a single memory channel. We also show a sensitivity analysis for a number of parameters: core count, channel count, metadata cache organizations, address mapping, etc.

Pin Traces	4 cores, filtered by 8MB LLC
Simulation ROB/width	64 entry/4-wide
MAC/parity/counter $	64KB shared by 4 cores
DDR3	Micron DDR3-1600 [27],
Baseline DRAM	64GB, 1 Channel, 16 ranks
Mem. Rd/Wr Queue	48/48 entries per channel
DRAM timing Parameters (DRAM cycles)	t_{RC} = 39, t_{RCD} = 11, t_{RAS} = 28, t_{FAW} = 20, t_{WR} = 12, t_{RP} = 11, t_{RTRS} = 2, t_{CAS} = 11, t_{RTP} = 6, t_{CCD} = 4, t_{WTR} = 6, t_{RRD} = 5, t_{REFI} = 7.8μs, t_{RFC} = 640 ns

TABLE III: *Simulator parameters.*

For the baseline, we assume 64 KB for a shared security/reliability metadata cache total for 4 cores; ITESP uses 16 KB metadata cache per core; we also perform a sensitivity analysis for the metadata cache size. Note that prior work has already shown that separate metadata caches work better than placing metadata in a larger LLC [34]. In the secure baseline (VAULT [40]), a 32 KB cache is used to store counters and integrity tree nodes, while a second 32 KB cache stores the most recently accessed MACs. In this configuration, reliability metadata is transferred along with data and stored in the 9th chip of an ECC DIMM. Baseline Synergy assumes a single 64 KB cache to store most-recently used counters and integrity tree nodes. When we augment Synergy with a parity cache, the metadata cache is split into two 32 KB caches, one for parity and one for counter/tree nodes. The parity cache is not



	Par./MAC $	Cnt $		Par./MAC $	Cnt $		Par./MAC $	Cnt $
VAULT	8KB	8KB	ITSYN+P$	8KB	8KB	Shared Par. +P$	8KB	8KB
SYNERGY	-	16KB	Shared Par.	-	16KB	ITESP	-	16KB

Fig. 8: *Execution time for the secure VAULT baseline, Vault with isolated trees and metadata caches (ITVAULT), VAULT+Synergy baseline (SYNERGY), VAULT+Synergy with isolation (ITSYNERGY), ITSYNERGY with a parity cache, ITSYNERGY with shared parity (no parity cache), ITSYNERGY with shared parity and a parity cache, and the proposed* ITESP*, all normalized to the non-secure baseline. Assumes 4 cores and 1 memory channel. The benchmarks are organized by the suite.*

SPEC2017		GAP	
Name	Working Set (MB)	Name	Working Set (MB)
perlbench	48	**bc**	12654
gcc	6425	**bfs**	8179
bwaves	10763	cc	6326
mcf	1760	sssp	1884
cactuBSSN	6476	**pr**	6530
namd	239	**tc**	9746
lbm	42	**NAS**	
omnetpp	3210	**bt**	2.6K
xalancbmk	156	**cg**	9K
cam4	168	ep	24
deepsjeng	6976	lu	2.7K
imagick	3245	**ua**	4.2K
fotonik3d	310	is	1K
roms	76	**mg**	15K
xz	7370	**sp**	2.7K
		ft	137
		dc	100

TABLE IV: *Benchmark specifications. The 15 most memory-intensive benchmarks are shown in bold font.*

filled by blocks read from memory; it simply stores 64-bit parities for recently written blocks; this helps coalesce multiple parity writes into a single parity block write when the block is evicted from the parity cache; this cache needs 8 valid bits per block to indicate dirty parity words per block. When a block is evicted from the parity cache, we use Masked Write Transfer (MWT) [16] to write only the updated portions back to the memory. In the proposed ITESP organization, a 16 KB metadata cache per enclave is used to store metadata blocks that include both counters and parity.

V. Results

In Section V-A, we first examine performance and energy improvements for a baseline that integrates the Synergy and ITESP techniques into an integrity tree modeled after VAULT [40], i.e., an integrity tree with variable arity (arity of 64 for leaf level, 32 for parent level, and 16 for grandparent level). We then discuss performance and energy for a baseline that integrates the Synergy and ITESP techniques into an integrity tree modeled with Morphable counters [33], i.e., a tree with even higher arity (64 and 128) and small local counters susceptible to high overflow rates.

A. ITESP *for VAULT and Synergy Baselines*

Figure 8 shows the execution time for the most relevant systems, all normalized against a non-secure baseline. For completeness, we show results for all 31 benchmarks, but for most of our discussion, we will report improvements for our 15 most memory-intensive benchmarks, indicated in Table IV. The integrity trees in this analysis are similar to VAULT, with arity of 64, 32, and 16 for the three lowest levels. To explain these results, Figure 9 shows the metadata overhead imposed by the most relevant models on every memory read and write. We confirm that similar to prior work, the Synergy baseline is 13.5% better than the VAULT baseline (Figure 8), with 20% lower metadata overhead (Figure 9). Note that the parity write traffic in baseline Synergy is high because it isn't cached.

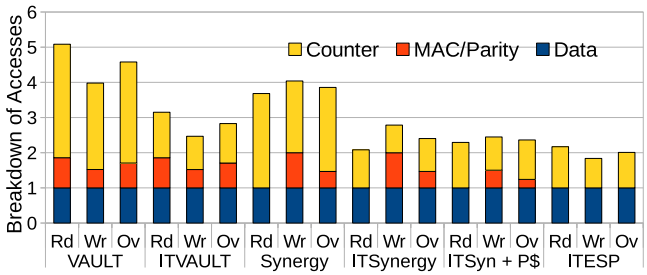


Fig. 9: *Breakdown of data+metadata accesses for each read and write operation. Averages are reported for the top-15 memory-intensive benchmarks.*

Adding isolation to both VAULT and Synergy has a significant performance impact, yielding performance improvements of 46% and 45% respectively. This is primarily because the metadata cache miss rate and metadata overhead are roughly halved (2.8 metadata blocks per data miss in Synergy is reduced to 1.4 with isolation, Figure 9) by avoiding inter-program interference. We observed that most of the benefit was because of tree isolation, i.e., it enabled higher-level tree nodes to capture metadata for a localized set of pages from one application, instead of scattered pages from multiple applications. Metadata cache partitioning had a very minor impact on cache hit rates, but is vital for leakage elimination.

The fifth bar in Figure 8 then incorporates a parity cache in Isolated Synergy to coalesce parity writes when spatial locality is observed in the write stream. We observe that such write coalescing improves performance by 3% because it reduces the parity write traffic by 49%. We then introduce parity sharing (the next two bars, without and with a parity cache). Unfortunately, parity sharing increases execution time because of the need to perform read-modify-writes on parity; even with a parity cache, execution time on average is similar to ITSYNERGY. Parity sharing does improve the effectiveness of the parity cache; on average, the hit rate of the parity cache improves from 45% to 60% with sharing across 16 blocks.

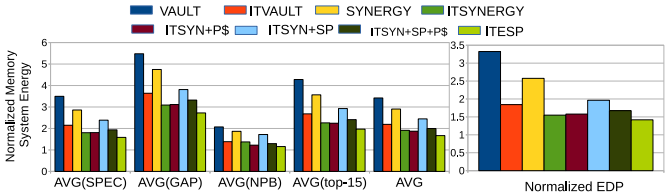


Fig. 10: *Normalized memory energy (on the left) and normalized average system energy delay product (EDP, on the right) for the same models described in Figure 8.*

Finally, we embed parity information into the tree with ITESP. This model yields performance that is 64% higher than the Synergy baseline (execution time reduction of 39%), 19% higher than ITSYNERGY, and 13% higher than ITSYNERGY with a parity cache. As shown in Figure 9, ITESP eliminates accesses to the separate MAC/parity data structure (0.46 per read/write in ITSYNERGY), but slightly increases accesses to the tree data structure (from 0.93 per read/write in ITSYNERGY to 1.0 in ITESP). Thus, every read/write memory operation in ITESP either requires (i) no additional memory accesses (if the leaf node is in the metadata cache), (ii) one additional memory access (if the leaf node is not in the metadata cache, but its parents are in the metadata cache), and (iii) more than one additional memory accesses (if leaf and its ancestors are not in the metadata cache).

Figure 10 shows memory energy results and system energy delay product. The memory energy reductions follow the same trend as the metadata traffic reductions. For the top-15 benchmarks, ITESP reduces memory energy by 45% and system EDP by 45%, relative to the Synergy baseline.

B. *Sensitivity Analysis*

Core Count

To understand the impact of executing a larger number of applications, Figure 12 summarizes the normalized execution times, memory energy, and system EDP for Synergy, and for ITESP, with 4 and 8 copies of the program. We execute the 4-core model with a single memory channel, while the 8-core model uses two memory channels. We observe that the baseline Synergy has a higher slowdown with higher core count even with more memory channels. This is primarily because of higher metadata cache misses from a greater degree of inter-program interference. The improvements from ITESP are therefore higher in the 8-core case. For the top-15 benchmarks, the performance improvement, memory energy reduction, and system EDP reduction go from 64%, 44%, 44% with 4 cores to 82%, 48%, and 73% with 8 cores, respectively.

Size of Metadata cache

Figure 13 depicts a similar sensitivity analysis for the metadata cache size per core. Larger metadata caches improve the key metrics for all configurations by similar amounts. With larger metadata caches, memory accesses are slightly lower, thus slightly reducing the benefits of ITESP. In terms of performance, the improvement with ITESP is 59% with 32 KB

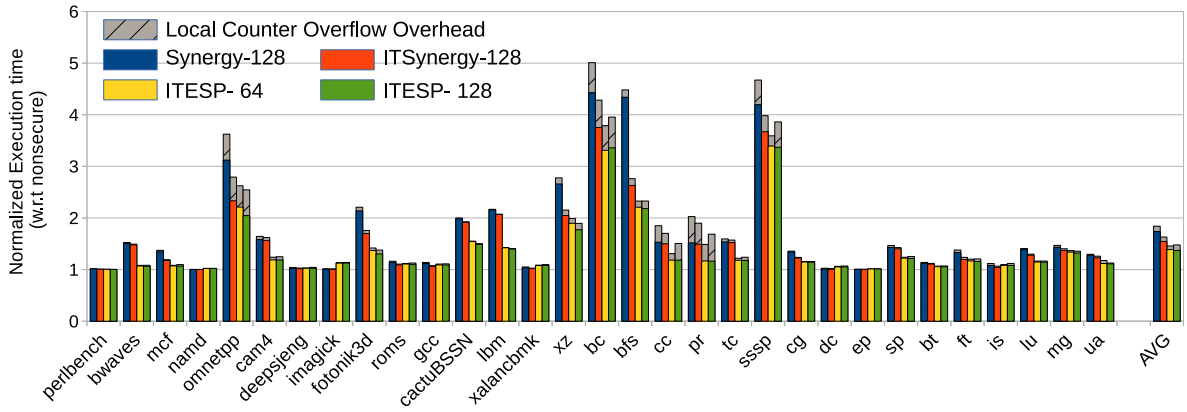


Fig. 11: *Normalized execution time (incl. local counter overflows) for Synergy and Morphable Counters (Synergy128), Synergy128 with Isolation, and* ITESP *with Morphable Counters (*ITESP *64 and* ITESP *128). Assumes 8 cores with 2 channels.*

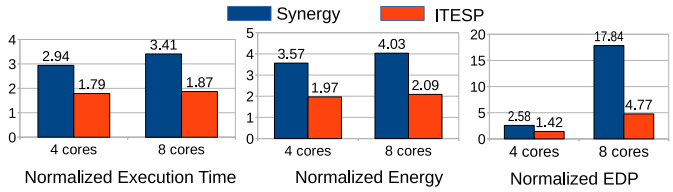


Fig. 12: *Execution time, memory energy, and system EDP for a 4-core model with 1 channel and an 8-core model with 2 channels, normalized against a non-secure baseline.*

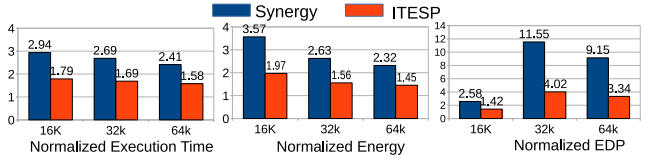


Fig. 13: *Execution time, memory energy, and system EDP for various metadata cache sizes, normalized against a non-secure baseline. The bars represent averages over top-15 memory-intensive benchmarks.*

metadata caches per core, and 52% with 64 KB metadata caches. Note that metadata caches in commercial systems are typically small; another perspective is that innovations like ITESP are helpful in achieving high performance levels with limited metadata cache space.

C. Address Mapping Policies

We next explore the design space of address mapping policies. Because parity is shared by blocks in different ranks, how consecutive blocks are interleaved can impact metadata cache miss rates and row buffer hit rates. Below, we identify address mapping policies that can balance the two. Figure 14 summarizes 4 relevant address mapping policies. Figure 15 summarizes the performance improvement, metadata cache miss rate, and row buffer hit rate for ITESP for each of these address mapping policies (for top-15 benchmarks). The performance improvement is relative to Synergy, with its best address mapping policy.

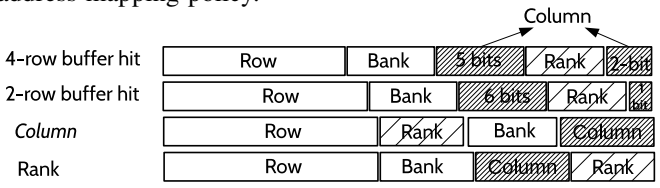


Fig. 14: *Address mapping policies for a 1-channel config.*

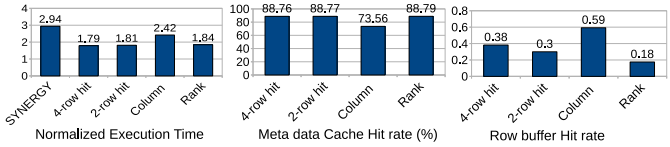


Fig. 15: *Impact of address mapping policies on performance, metadata cache miss rate, and row buffer hit rate (assuming 4 cores and 1 channel).*

The first policy, *Column*, places consecutive cache lines in a single row buffer. It therefore yields a high row buffer hit rate. But because these consecutive cache lines are mapped to different shared parity blocks in ITESP, they suffer from a high metadata cache miss rate, i.e., the address mapping policy that was best for Synergy is highly sub-optimal for ITESP. The *Rank* address mapping policy places consecutive cache lines in different ranks; it thus lowers metadata cache miss rate in ITESP, but also offers a very low row buffer hit rate. To alleviate these problems, we introduce the *2-row buffer hit* and *4-row buffer hit* policies. In the latter, 4 consecutive cache lines are placed in the same row buffer, thus promoting row buffer hit rates. Even though these 4 consecutive cache lines map to different shared parities, because a leaf node has 4 different shared parities, they can map to a single leaf node. Such a mapping therefore promotes row buffer hit rates without compromising metadata cache miss rate.

D. ITESP *with Morphable Counter Baseline*

Finally, we show that ITESP is also compatible with innovations [33] that exploit counter value locality to increase tree arity. Recall the three configurations described in Figure 7, one (SYN128) that resembles a Synergy-like baseline with Morphable Counters, and two (ITESP 64 and ITESP 128) that integrate ITESP and Morphable Counters. Figure 11 quantifies the execution time impacts of these models for an 8-core 2-channel configuration. These results also include the overheads incurred when dealing with local counter overflows, given the small sizes for local counters. This is estimated with a separate long Pin-based simulation that does not model per-cycle effects, but models counter values. The overflow rate is directly related to the local counter size – 2 bits for ITESP 128, 3 bits for Synergy, and 5 bits for ITESP 64. We see that ITESP 64 is the best organization by a small margin, i.e., its lower overflow rate overcomes its lower metadata cacheability. It achieves an improvement of 27% over Synergy, 12.4% over ITSynergy, and 1.4% over ITESP 128. With higher arity trees, most metadata cache misses are localized to the leaf nodes of the tree. This reduces the benefit from isolation (which primarily targets interference in higher levels of the tree), but increases the benefit from embedded shared parity (which targets the organization of the leaf node).

VI. Related Work

Memory Integrity Verification. We have already discussed state-of-the-art integrity verification techniques in Section II: BMT, MEE, VAULT, Synergy, Morphable Counters. The work of Lehman et al. [22] analyzes different caching strategies for integrity metadata. Two works, InvisiMem [2] and ObfusMem [4], show how memory devices with logic capabilities can lower the overheads for both integrity verification and oblivious RAM.

Memory Reliability. Multiple works, e.g., [50], [19], [17], [41], have constructed codes for chipkill that improve performance, energy, and capacity metrics. Some of these works [41] use parity and RAID-like approaches for error correction. ECC-parity [18] shares the ECC bits among different channels to reduce the power and capacity overhead of a reliable memory system. Multi-ECC [17] provides chipkill for an ECC DIMM by leveraging a shared checksum.

Unified Reliability and Integrity. In addition to Synergy, IVEC [13] offers a combined solution for both integrity and chipkill. Unlike Synergy that exploits an ECC DIMM, IVEC supports chipkill for non-ECC DIMMs. Integrity trees for persistent memory systems must correctly recover metadata after crashes – Osiris [49] and Anubis [51] show how encryption counters and integrity tree nodes can be recovered at low cost, e.g., using ECC to sanity check encryption counters.

VII. Conclusions

This work first isolates each enclave's integrity tree to reduce negative interference and eliminate two potential side channels. We then build on Synergy by observing that parities can be shared by multiple blocks, thus bringing parity overhead on par with counter overhead. This enables placing both parity and counters in a single node of the integrity tree. Isolation improves performance of Synergy by 39% and the unified data structure boosts this improvement to 64%. Parity sharing has a negligible impact on reliability, primarily causing new DUEs in the unlikely event of independent memory chip failure in different ranks within a short window. The proposed approach does not require write masking and therefore offers broader system compatibility.

VIII. Acknowledgments

We thank the anonymous reviewers for many helpful suggestions. This work was supported in parts by NSF grant CNS-1718834 and Intel. We thank Anton Burtsev and Karl Taht for their help with timing measurements on SGX.

References

[1] "Micron System Power Calculator," http://www.micron.com/products/support/power-calc.

[2] S. Aga and S. Narayanasamy, "InvisiMem: Smart Memory for Trusted Computing," in *International Symposium on Computer Architecture*, 2017.

[3] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell *et al.*, "SCONE: Secure Linux Containers with Intel SGX." in *OSDI*, 2016, pp. 689–703.

[4] A. Awad, Y. Wang, D. Shands, and Y. Solihin, "ObfusMem: A Low-Overhead Access Obfuscation for Trusted Memories," in *International Symposium on Computer Architecture*, 2017.

[5] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, and V. V. andd S. Weeretunga, "The NAS Parallel Benchmarks," *International Journal of Supercomputer Applications*, vol. 5, no. 3, pp. 63–73, Fall 1994.

[6] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP Benchmark Suite," *arXiv*, 2015.

[7] J. Bucek, K. Lange, and J. V. Kistowski, "SPEC CPU2017: Next-Generation Compute Benchmark," in *ACM/SPEC International Conference on Performance Engineering*, 2018.

[8] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, "USIMM: the Utah SImulated Memory Module," University of Utah, Tech. Rep., 2012, UUCS-12-002.

[9] V. Costan and S. Devadas, "Intel SGX Explained," 2016, https://eprint.iacr.org/2016/086.pdf.

[10] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev, "BranchScope: A New Side-Channel Attack on Directional Branch Predictor," in *Proceedings of ASPLOS*, 2018.

[11] A. Ferraiuolo, Y. Wang, D. Zhang, A. C. Myers, and G. E. Suh, "Lattice Priority Scheduling: Low-Overhead Timing Channel Protection for a Shared Memory Controller," in *Proceedings of HPCA*, 2016.

[12] S. Gueron, "A Memory Encryption Engine Suitable for General Purpose Processors," in *Proceedings of IACR*, 2016.

[13] R. Huang and G. Suh, "IVEC: Off-Chip Memory Integrity Protection for Both Security and Reliability," in *Proceedings of ISCA*, 2010.

[14] C. Hunger, M. Kazdagli, A. Rawat, S. Vishwanath, A. Dimakis, and M. Tiwari, "Understanding Contention-driven Covert Channels and Using Them for Defense," in *Proceedings of HPCA*, 2015.

[15] Intel, "HBM Interface Intel FPGA IP User Guide," 2019. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/documentation/mhi1462215825912.html

[16] JEDEC, "Masked Write Transfer," https://www.jedec.org/standards-documents/dictionary/terms/masked-write-transfer-mwt.

[17] X. Jian, H. Duwe, J. Sartori, V. Sridharan, and R. Kumar, "Low-Power, Low-storage-overhead Chipkill Correct via Multi-line Error Correction," in *Proceedings of SC*, 2013.

[18] X. Jian and R. Kumar, "ECC Parity: A Technique for Efficient Memory Error Resilience for Multi-Channel Memory Systems," in *Proceedings of SC*, 2014.

[19] J. Kim, M. Sullivan, and M. Erez, "Bamboo ECC: Strong, Safe, and Flexible Codes for Reliable Computer Memory," in *the Proceedings of HPCA-15*, February 2015.

[20] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," 2018, https://spectreattack.com/spectre.pdf.

[21] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "RAMBleed: Reading Bits in Memory Without Accessing Them," in *41st IEEE Symposium on Security and Privacy (S&P)*, 2020.

[22] T. Lehman, A. Hilton, and B. Lee, "MAPS: Understanding Metadata Access Patterns in Secure Memory," in *Proceedings of ISPASS*, 2018.

[23] T. S. Lehman, A. D. Hilton, and B. C. Lee, "PoisonIvy: Safe Speculation for Secure Memory," in *Proceedings of MICRO*, 2016.

[24] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of PLDI*, 2005.

[25] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar, "Innovative Instructions and Software Model for Isolated Execution," in *Proceedings of HASP Workshop, in conjunction with ISCA-40*, 2013.

[26] Micron, "DDR4 SDRAM RDIMM," 2013, product datasheet, https://www.micron.com/-/media/client/global/documents/products/data-sheet/modules/parity_rdimm/asf9c512x72pz.pdf.

[27] Micron Technology Inc., "Ddr3 sdram part mt41j256m8," 2006.

[28] K. Nguyen, "Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family," 2016, https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology.

[29] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: ExitLess OS Services for SGX Enclaves," in *EuroSys*, 2017, pp. 238–253.

[30] M. Qureshi, "CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping," in *Proceedings of MICRO*, 2018.

[31] J. Robertson and M. Riley, "The Big Hack: How China Used a Tiny Chip to Infiltrate U.S. Companies," 2018, https://tinyurl.com/ycywjdmo.

[32] B. Rogers, S. Chhabra, Y. Solihin, and M. Prvulovic, "Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly," in *Proceedings of MICRO*, 2007.

[33] 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 *Proceedings of MICRO*, 2018.

[34] G. Saileshwar, P. Nair, P. Ramrakhyani, W. Elsasser, and M. Qureshi, "SYNERGY: Rethinking Secure-Memory Design for Error-Correcting Memories," in *Proceedings of HPCA*, 2018.

[35] B. Schroeder, E. Pinheiro, and W. D. Weber, "DRAM Errors in the Wild: A Large-Scale Field Study," in *Proceedings of SIGMETRICS*, 2009.

[36] M. Seaborn and T. Dullien, "Exploiting the DRAM Row Hammer Bug to Gain Kernel Privileges," in *Black Hat*, 2015.

[37] A. Shafiee, A. Gundu, M. Shevgoor, R. Balasubramonian, and M. Tiwari, "Avoiding Information Leakage in the Memory Controller with Fixed Service Policies," in *Proceedings of MICRO*, 2015.

[38] V. Sridharan and D. Liberty, "A Study of DRAM Failures in the Field," in *Proceedings of SC*, 2013.

[39] V. Sridharan, N.DeBardeleben, S. Blanchard, K. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory Errors in Memory Systems: The Good, The Bad, and The Ugly," in *Proceedings of ASPLOS*, 2015.

[40] M. Taassori, A. Shafiee, and R. Balasubramonian, "VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures," in *Proceedings of ASPLOS*, 2018.

[41] A. N. Udipi, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. Jouppi, "LOT-ECC: Localized and Tiered Reliability Mechanisms for Commodity Memory Systems," in *Proceedings of ISCA*, 2012.

[42] A. Vuong, A. Shafiee, M. Taassori, and R. Balasubramonian, "An MLP-Aware Leakage-Free Memory Controller," in *Proceedings of HASP Workshop, in conjunction with ISCA-45*, 2018.

[43] Y. Wang, B. Wu, and G. Suh, "Secure Dynamic Memory Scheduling Against Timing Channel Attacks," in *Proceedings of HPCA*, 2017.

[44] Y. Wang, A. Ferraiuolo, and G. E. Suh, "Timing Channel Protection for a Shared Memory Controller," in *HPCA*, 2014.

[45] Z. Wang and R. B. Lee, "New Cache Designs for Thwarting Software Cache-based Side Channel Attacks," in *Proceedings of ISCA*, 2007.

[46] ——, "A Novel Cache Architecture with Enhanced Performance and Security," in *Proceedings of MICRO*, 2008, pp. 83–93.

[47] Xilinx, "DDR3 ECC with Data Mask," 2013. [Online]. Available: https://forums.xilinx.com/t5/Other-FPGA-Architectures/ddr3-ecc-with-data-mask/td-p/570028

[48] M. Yan, J.-Y. Wen, C. Fletcher, and J. Torrellas, "SecDir: A Secure Directory to Defeat Directory Side-Channel Attacks," in *Proceedings of ISCA*, 2019.

[49] M. Ye, C. Hughes, and A. Awad, "Osiris: A Low-Cost Mechanism to Enable Restoration of Secure Non-Volatile Memories," in *Proceedings of MICRO*, 2018.

[50] D. Yoon and M. Erez, "Virtualized and Flexible ECC for Main Memory," in *Proceedings of ASPLOS*, 2010.

[51] K. A. Zubair and A. Awad, " Anubis: Ultra-Low Overhead and Recovery Time for Secure Non-Volatile Memories," in *Proceedings of ISCA*, 2019.